Trabajo Fin de Grado
Ingeniería de Telecomunicación

Application of deep learning methods to the mitigation of nonlinear effects in communication systems

Autor: Luis Álvarez López

Tutor: Juan Antonio Becerra González

Dpto. Teoría de la Señal y Comunicaciones
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2019

Trabajo Fin de Grado
Ingeniería de Telecomunicación

Application of deep learning methods to the mitigation of nonlinear effects in communication systems

Mitigación de efectos no lineales en sistemas de comunicación basado en técnicas de aprendizaje profundo

Autor:
Luis Álvarez López

Tutor:
Juan Antonio Becerra González
Profesor Sustituto Interino

Dpto. Teoría de la Señal y Comunicaciones
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2019

Trabajo Fin de Grado:	Application of deep learning methods to the mitigation of nonlinear effects in communication systems

Autor:	Luis Álvarez López
Tutor:	Juan Antonio Becerra González

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

Agradecimientos

Me gustaría empezar esta sección agradeciendo a todos esos compañeros que hacen que el día a día en la facultad no sea una rutina. Con ellos, todo es más fácil. Solo ellos saben lo que significa pasar estos cuatro años juntos, y solo por ellos, los repetiría. Gracias, especialmente, a Rafa por estar siempre ahí, motivándome y ayudándome, a Juan con sus *días tristes* escuchando *The Scientist*, a Antonio por tener la respuesta a todo, a Leo, David, Gonzalo... Ha sido un placer teneros como compañeros y me considero muy afortunado de haber coincidido con vosotros.

Para continuar, me siento en el deber de agradecer a todos los profesores de los que he recibido alguna práctica o clase. Gracias por enseñarme a aprender por mí mismo, por apoyarme por el duro camino de convertirme en un ingeniero. Especialmente, me gustaría agradecer a Juan Antonio Becerra, tutor de este trabajo, por demostrar ser una persona (y un profesor) diferente desde el primer día en aquella práctica de Comunicaciones Digitales. Gracias por enseñarme a mantener las neuronas activas durante este *entrenamiento* que ha supuesto mi iniciación en la investigación. Gracias por saber motivarme y educarme mucho más allá de lo que está dentro de tu deber como profesor.

Para terminar, gracias a mi familia. Muchas gracias a mis hermanos, Susi y Nano por marcarme el camino con vuestro ejemplo. Cabe destacar también la inestimable ayuda de Tere y Loli, con sus llamadas, su apoyo incondicional y sus comidas. Y, por último, gracias a mis padres, porque sois, desde luego, las personas más influyentes en mi vida, un ejemplo y una inspiración. Como diría Umberto Eco: *"Creo que aquello en lo que nos convertimos depende de lo que nuestros padres nos enseñan en pequeños momentos, cuando no están intentando enseñarnos. Estamos hechos de pequeños fragmentos de sabiduría."* Me faltarían renglones en este trabajo para agradeceros vuestro apoyo como de verdad os merecéis. Conformándome con este parrafo, me gustaría daros las gracias por enseñarme a luchar y trabajar por lo que quiero, incluso en los días más difíciles, y enseñarme el valor de la constancia. Os quiero.

Luis Álvarez López
Sevilla, 2019

Resumen

En este trabajo se presentan tanto el desarrollo matemático e implementación de una red neural compleja como la aplicación del algoritmo *Random Forest* al problema de cálculo de importancias de los regresores utilizados en el modelado, basado en series de Voterra, de un amplificador de potencia. En primer lugar, se ha realizado una descripción de la situación actual de los algoritmos de aprendizaje profundo. En esta descripción se han ofrecido detalles sobre su origen, aplicaciones actuales y evolución. Seguidamente, se presenta una vista en profundidad del campo de las redes neuronales, que concluye con el desarrollo matemático de una red neuronal compleja. A continuación, se encuentra una introducción teórica a los amplificadores operacionales, el origen de su comportamiento no lineal y las diferentes soluciones que han sido estudiadas. La parte experimental se basa en la aplicación del algoritmo de Random Forest al cálculo de un vector de importancias de regresores, el cual podría ser usado para reducir un modelo de Volterra para ganar eficiencia energética sin perder demasiada precisión. Para la validación del algoritmo se ha utilizado una señal 5G-NR de 30 MHz, mientras que el amplificador operacional fue fijado a un punto de operación de +27.5 dBm de potencia media de salida, correspondiente a 1.2 dB de compresión de ganancia. Los resultados experimentales muestran las ventajas de este algoritmo respecto a otros.

Abstract

In this project, both a mathematical development of a Complex Valued Neural Network (CVNN) and the application of a Random Forest algorithm to Volverra regressors importance estimation are presented. Firstly, an overview about machine and deep learning algorithms is presented, where details about the origin, actual application and future are discussed. After this, an insight into the field of artificial neural networks (ANNs) is offered. This insight concludes with an explanation of the mathematical development of a CVNN. Following, a theoretical introduction to power amplifiers (PAs), the origin of their nonlinearities and the different solutions that have been approached is presented. Finally, experimental part is based on the application of Random Forest to Volterra regressors importance calculation, from which pruning decisions can be made. The validation of the presented technique was executed over a 30 MHz 5G-NR signal. The PA operating point was fixed to +27.5 dBm of average output power which corresponds to 1.2 dB of gain compression. Experimental results are illustrated with figures which show the advantages of this algorithm against others.

Índice Abreviado

Contents

1 Introduction

There's a time when a man needs to fight, and a time when he needs to accept that his destiny is lost, the ship has sailed and only a fool would continue. Truth is, I've always been a fool.

Edward Bloom in *Big Fish*, 2003

The evolution of technology have led to a situation where almost every industry needs a wireless communication system. Nowadays, communications are characterized by digitally modulated signals with an efficient usage of the spectral bandwidth. These practices allow a wide range of different signals to coexist in the same channel without jamming problems. In addition to this variety, clients demand higher data rates without losing technical features such as capacity, flexibility and reliability. However, every communication company tries to offer its services to the highest amount of clients with the lowest expenses. Besides that, climate change concerns have increased the focus on energy wastage and these are the reasons why power efficiency has become a critical topic of interest.

Recent studies have estimated that the communication industry is responsible for around a 10% of the total power consumption [1]. Radio equipment represents the most energy consuming field, specially power amplifiers (PAs) which consuming approximately a 70% of the total input power [2]. A power amplifier is an active device. This fact means that it needs a power supply, so it obeys the energy conservation law. Therefore, its efficiency is better when the output power is closer to the power that it is supplied with. Due to their considerable percentage of the total power consumption of a wireless communication system, optimization of PAs efficiency has recently been a case of study.

As it has already been commented, PAs are limited by the energy conservation law. This characteristic provokes that, although they are more efficient when the output power is close to the supply power, they also start working nonlinearly. This area of operation is known as the *saturation zone*, because *clipping* effects tend to appear. A PA presents a clipping behavior when it tries to produce an output power which is higher than its supply power. Because of physics, this is impossible and that is why the output power stops increasing no matter how much the input power continues getting higher. Nonlinearities trigger injurious effects in the output signal, since its spectral bandwidth is affected. They are responsible of different kind of distortion, being out-of-band distortion especially harmful since it causes adjacent channel interference. Since, channels are carefully distributed by law, this kind of problems need to be taken seriously into account when a communication scheme is designed.

Recent modulation techniques such as Orthogonal Frequency Division Multiplexing (OFDM) are characterized by a high Peak-to-Average Power Ratio (PAPR) [3]. This indicator is computed as the relation between the maximum power of a sample in a symbol divided by the average power of that symbol. Despite this drawback, its numerous advantages such as high bit rate, strong immunity to multipath and high spectral efficiency make this modulation method very interesting and that is why it is widely used in these days. Combining the usage of a high PAPR modulation with a PA working at its saturation zone results in nonlinearities. This conflict of interests is known as the linearity-efficiency trade-off.

Since this problem was firstly encountered, different linearization approaches have been presented [4]. Interest in linearization started in the 1920s, where two of the three techniques that have been used until now where introduced: feedforward and feedback. In this work, we will be focus on predistortion, which is the latest linearization solution and the most used nowadays. It was conceived as an analog technique but during the 1990s decade, digital signals started to dominate the industry so that the method also became digital.

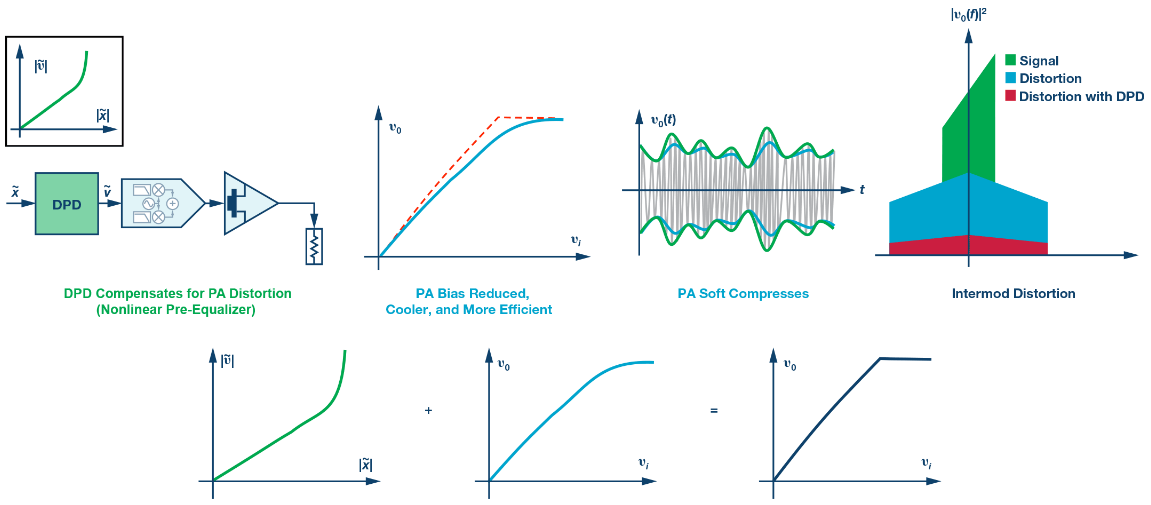

Figure 1.1 Digital predistortion (DPD) scheme for PA linearization [5].

As it is illustrated in Fig. 1.1, DPD is based on distorting the input signal before it is fed to the power amplifier. It compensates nonlinearities before they appear, working in cascade with the power amplifier. As it shown in the figure a DPD block before a PA at its saturation point are able to achieve a significant linear behavior. Therefore, spectral distortion is reduced so that linearity and efficiency can be achieved at the same time.

In order to use DPD, an accurate model of the power amplifier behavior (both its linear and nonlinear behaviors) is required. At first, lookup tables (LUTs) were used to identify the input-output signal relationship of the power amplifier and the response is inverted to remove nonlinearities. However, this is a very simple workaround which does not take into account memory effects. Volterra series models were presented as an improvement of LUTs. They belong to the field of behavioral modeling and they are able to reproduce any nonlinear behavior even with memory effects. They are based on a set of regressors of which importance needs to be computed as a coefficient or *kernel* depending on the behavior which is being modeled. Volterra series pruning and computational complexity have been interesting challenges in the recent industry [**?**]. At this point, deep learning methods have appeared as interesting approaches to this situation.

Chapter 2 of this work presents an overview of Artificial Intelligence (AI) different methods and applications. In Chapter 3, Artificial Neural Networks (ANNs) inner working is deeply explained and their main applications are discussed. A Complex Valued Neural Network (CVNN) is also mathematically developed in this chapter. Eventually, Chapter 4 offers an insight to power amplifiers, why and how nonlinearities. In addition to this, different approaches of Volterra series development are offered in order to ease the understanding. Various models derived from this series are also introduced in Chapter 4. Chapter 5 presents the application of a Random Forest technique to the estimation of Volterra regressors importance and results are offered to illustrate its advantages.

2 Artificial Intelligence Overview

What we want is a machine that can learn from experience.

Alan Turing

Deep learning is a technology that imitates the way how human brain processes information and find patterns. It is based on machine learning algorithms and requires large amounts of input-output data in order to get accurate results. Nowadays, this breakthrough technology can be applied to different fields of study. It has already been used on applications such as self-driving cars, health care, recommendation systems, voice or image recognition and even at fraud detection. Furthermore, new applications are found every day thanks to the flexibility of deep learning [6]. Before talking about its origin, it is important to define clearly the concepts of artificial intelligence (AI), machine learning and deep learning, since they are usually mixed and confused. AI is the widest field and it includes the rest of them. AI is defined as the general concept or idea of creating machines that are able to behave in a manner that is similar to the human behavior. Machine learning is a subset of AI, since it is a group of mathematical and statistical algorithms that make it possible for machines to learn [7]. Deep learning, finally, is a particular field of machine learning, it can be understood as the next step. It is based in *deep* structures that use machine learning algorithms repeatedly in order to make this behavior closer to the human brain [8].

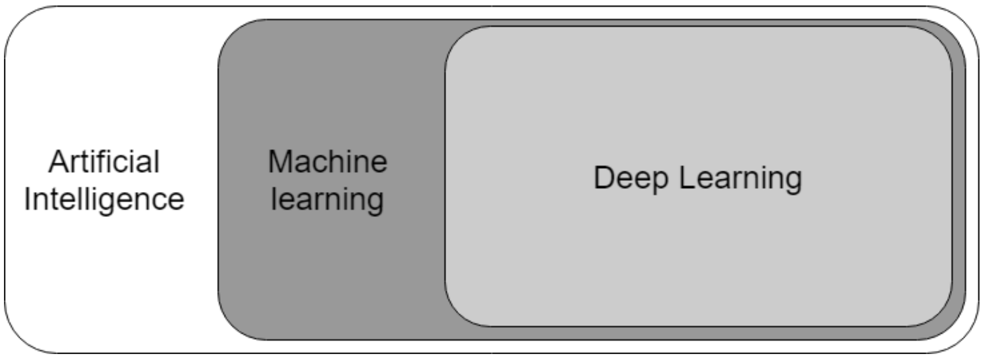

Figure 2.1 Relation between AI, machine and deep learning..

2.1 Origin

In order to understand the importance of deep learning in the current scientific field it is interesting to look backwards and get to know its first steps. The study of the human brain behavior was first faced by philosophers. Aristotle introduced the idea of understanding the human brain behavior through a few basic concepts and functionalities combined between each other. After that, this theory named *associationism* [9] became popular but it was not until 1873 when Alexander Bain first attempted to model the human brain storage ability with the definition of *neural groupings* [10]. Bain's diagram of a neural grouping is shown in Fig. 2.2. Following the associationism theory, neural groupings were based on a structure of basic cells linked between each other and they are considered as the very first artificial neural networks. Bain highlighted the computational flexibly of neural groupings against the traditional prestored memories. He talked about the connections between basic cells and how these links could get stronger or weaker depending on the experience, introducing the idea of the actual update equation used in deep learning.

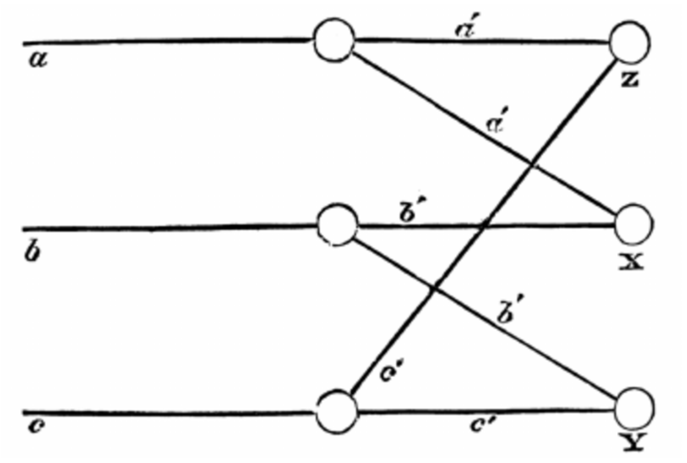

Figure 2.2 Bain neural grouping schema [10].

Nowadays, ANNs represent connections with the usage of weights and the experience that Bain talked about is now known as the training stage. Another important milestone during the evolution of deep learning was the statement of the Hebbian Learning Rule in 1949. Donald O. Hebb stated in his book The Organization of Behavior [11] that *"neurons that fire together, wire together"*, which can be expressed as

$$\Delta w_i = \eta x_i y, \tag{2.1}$$

where Δw_i refers to the increase of the weight in neuron i, η is the learning rate and y the output of the neural grouping. Despite the lack of citation, it seems likely that Hebb used Bain's ideas of weakening or strenghtening of connections between cells. Although the Hebbian rule is the basis of the update equation (2.1), it has a major drawback. Hebb stated that the relationship between two units should be continuously increased as long as they are firing together. However, this idea leads to an exponential increase of weights. To solve this problem Erkki Oja introduced a normalization term in the equation [13], defining Oja's rule as

$$\Delta w_i = \eta y(x_i - y w_i). \tag{2.2}$$

The next step in the evolution of this technology was the McCulloch-Pitts (MCP) neuron, developed by Walter Pitts and Warren McCulloch. It was a primary perceptron with static weights that had to be manually calculated and it was based on the step activation function. The success of the MCP model encouraged Frank Rosenblatt to use, in 1958, the Hebbian Learning rule with a perceptron in order to imitate the human eye behavior. This is considered as the very first neural network, although Rosenblatt's idea was more related with an electronic hardware than a software algorithm. Finally, in 1965 the mathematician Alexey Ivakhnenko developed the first working computer based neural network. He created a 8-layer perceptron and was able to demonstrate its ability of learning in 1971. Other techniques and concepts such as *backpropagation* have been introduced in order to extend or develop deep learning during the last years of the 19th century. Although applications of deep learning never stopped growing during this time, this growth cannot be compared to the one that deep learning technology is experimenting during the last decade. The reason for this remains on the technology that is required to get the most of deep learning. The essential requirements of deep learning are large datasets and computational capacity. Thanks to Moore's Law, both of them have experimented a major growth themselves, allowing deep learning to develop without constraints in this sense. Figure 2.3 illustrates the increasing of computational power through time until 2015. The *#500* line shows the 500 most powerful commercially available computer systems known to reference [12].

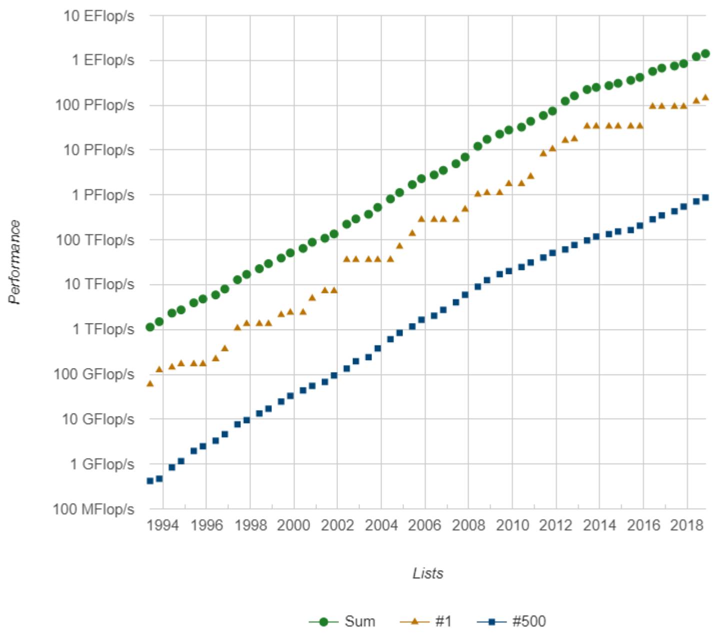

Figure 2.3 Computational speed evolution [12].

2.2 Classification

This evolution has led to the emergence of categories in deep learning depending on the concepts they are based on and the applications they are used for. There are three main branches of deep learning: supervised, unsupervised and reinforcement learning [6].

2.2.1 Supervised Learning

It is the most used. It requires a labeled dataset with the correct outputs. It can be compared to the way a child learns, through success and error. It is also called predictive learning, since its main target is to produce accurate outputs of a given input dataset. During this training, a cost function is used in order to minimize the differences between the desired output and the real one. It is often used for regression or classification tasks. Depending on its application and functionalities, this branch can be divided in three models:

Artificial Neural Networks

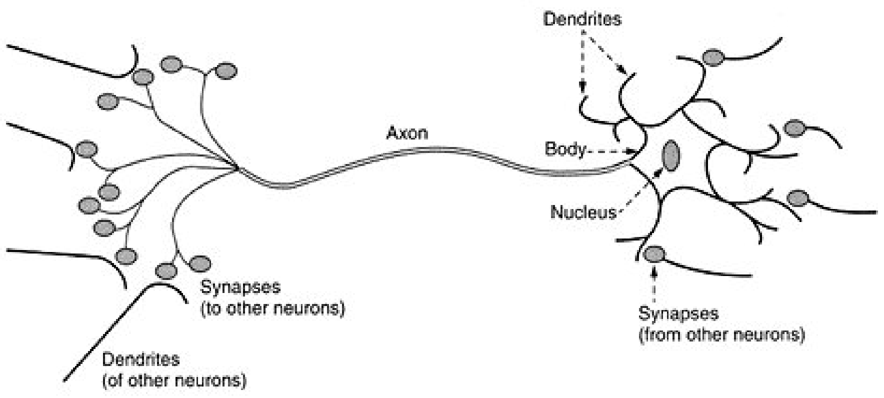

Figure 2.4 Biological neuron model [14].

Artificial neural networks (ANNs) are generally used as the basic structure [15]. Other types of deep learning techniques have emerged from variations of ANNs. They are constructed by layers of neurons connected between each other and they are helpful on finding and learning about nonlinear behaviors [16].

Computational model of a neuron was inspired by its biological model shown in Figure 2.4. Further study of this supervised learning model is provided in Chapter 3.

Convolutional Neural Nerworks

Convolutional neural networks (CNNs) are an extension of artificial neural networks in the computer vision field [17]. They are able to find features in a set of input images and use them to solve classification or recognition tasks. This ability make CNNs very interesting to apply in fields such as action or human pose recognition and scene labelling [18]. CNNs are based on the idea of preprocessing the input images and feed them into an ANN. This preprocessing converts the input image, which is a matrix, to a flat vector and it is accomplished by a sequence of layers. First, the input image matrix goes into a convolutional layer. The convolution is made between the image matrix and adjustable filter. A filter is a matrix, usually of size 2x2 or 3x3 that tries to find specific features or patterns. Hence, this process enables the feature detection and it is resistant to translation effects so it makes it easier to find the patterns regardless of the perspective of the image [17]. The output of the convolutional layer is known as feature map. Fig. 2.5 summarizes the structure of a CNN.

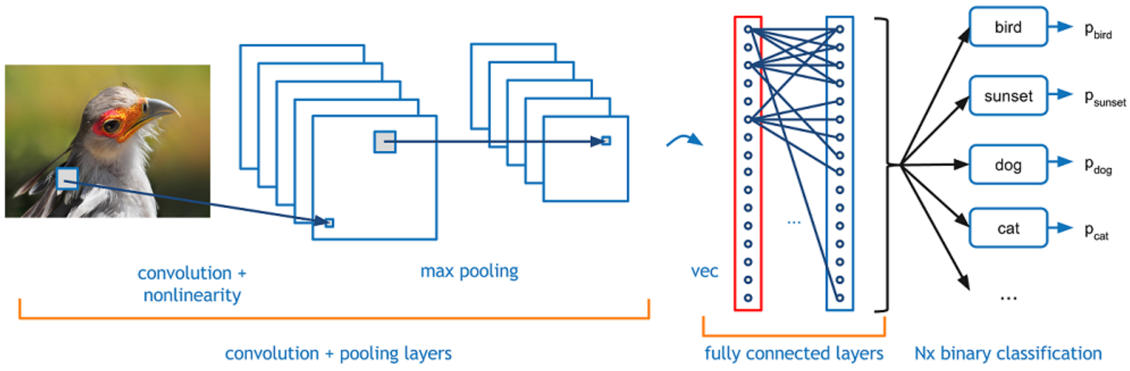

Figure 2.5 Structure of convolutional neural network for a classification task [19].

This map is introduced to a pooling layer. As the previous layer, this layer also reduces the size of the matrix we are using, which will reduce the computation requirements of the training stage. The principal target of this layer is to identify the most dominant features. After this, a flattening layer could be used to transform the obtained matrix into a single dimensional vector and use this vector as the input to what is known as a fully connected layer, an ANN. However, the processes of convolution and pooling are usually repeated a few times before the result is introduced to the fully connected layer.

Recurrent Neural Networks

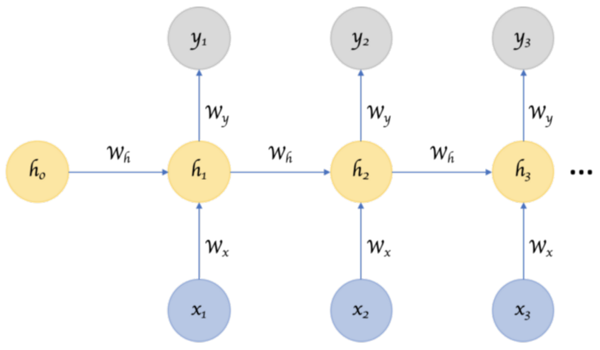

Figure 2.6 Structure of recurrent neural network [20].

Recurrent neural networks (RNNs) add a temporary sense to the traditional ANN structure. It is not an uncommon situation to find a single input sample w which has a correlation with others in the past and affects the ones in the future. A simple ANN would not consider this fact in order to map outputs to this inputs. RNNs try to add this memory functionality by considering not only the current input sample but also a hidden state that depends on previous outputs as it is shown in Fig. 2.6. This concept of memory allows RNNs to find correlations between previous inputs and the actual one that could not be found with an ANN. As an example, if we try to translate a sentence from one language to another, we will realize that one word in a language can affect the rest of the translation. RNNs have been proven to be powerful on tasks like this one, and that is why they are dominant in Natural Language Processing (NLP) nowadays. However, this dependency between the current output and the previous ones provokes the error to propagate forward and accumulate. That is why RNNs are not considered to work practically. Long-Short Term Memory (LSTMs) have recently emerged as a variant of RNNs without this problem. The problem is solved by using not only weights to connect the hidden state to the new input but also functions such as sigmoid and hyperbolic tangent in order to limit the influence of previous states. LSTMs are also better than RNNs at remembering previous states and avoiding the gradient to vanish. A general structure of an LSTM layer is reported in Fig. 2.7.

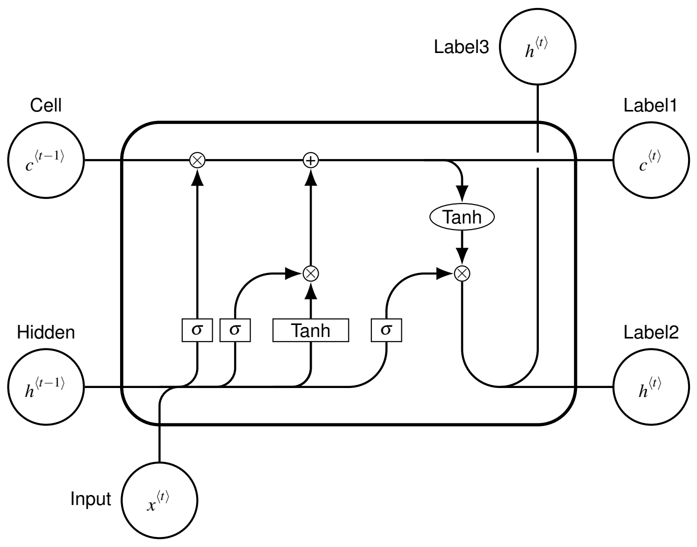

Figure 2.7 LSTM layer with the operations involved [21].

2.2.2 Reinforcement Learning

Instead of working with a cost function, it works with a reinforcement function to maximize through the interaction between the model and the environment. Reinforcement learning models do not have this requirement thanks to their direct interaction with the environment, so no feature engineering is needed. During an iteration, a policy function, which depends on the state of the context, is used to decide the action that the agent will perform [22]. Actions interact with the environment, hence they lead to a state change. After that, a scalar reward is computed with a reward function. The reward, state and actions will help the agent to determine its policy for the new iteration, and that is how the learning is achieved. A simple schema of the concepts under a reinforcement learning algorithm iteration is shown in Fig. 2.8.

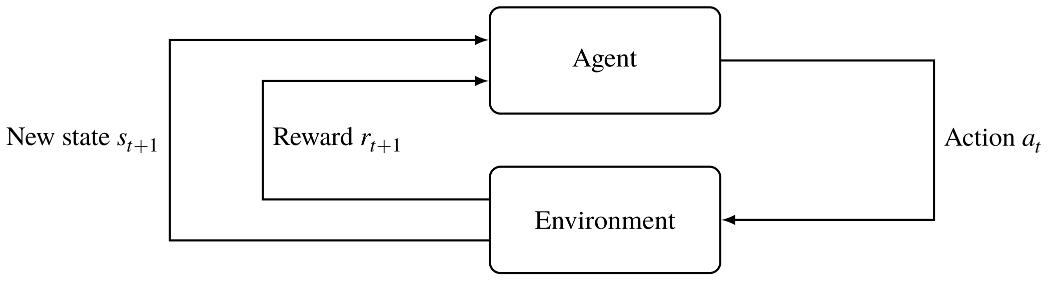

Figure 2.8 Reinforcement learning conceptual diagram [23].

2.2.3 Unsupervised Learning

In unsupervised learning, the input dataset is trained without a cost or a reward function. There are not desired outputs to obtain . The main goal is to find patterns in the data that allow to group it in clusters or compress it. A common unsupervised learning method is called Principal Component Analysis (PCA). This method to find linear combinations between a given data in order to summarize most of the information [24]. Auto-encoders are also used for this purpose, they are ANNs that find efficient data coding in order to compress the information used as input. Clustering has also emerged as useful application of Unsupervised Learning. It is based on finding groups in dataset that share things in common. It usually does it by reducing the Euclidean distance, but there are lots of variances. The results of the application of clustering techniques are shown in Fig. 2.9.

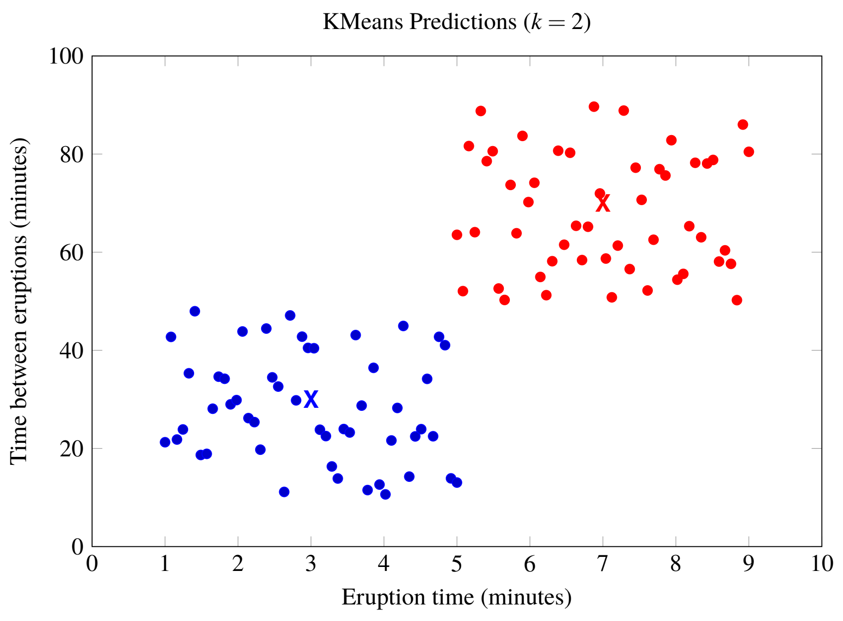

Figure 2.9 Euclidean clustering diagram on classification of breath tumors.

After reading the whole classification of deep learning methods, it may be confusing for the reader to understand when it is better to apply a specific model. Although some of this techniques have been highlighted as the most popular, it is important to understand that this fact does not mean that they are suitable for every problem that we may encounter. As a matter of fact, the first step of applying a deep learning technique to a certain problem should be getting a detailed knowledge about the problem involved and a thorough search of the most suitable AI technique for that case. An illustration that may make this task easier is presented in Fig. 2.10.

2.3 Evolution

Having already done an introduction about the most dominant deep learning techniques nowadays, it is worth mentioning how fast this technology is emerging. Scientific papers have been increasing on this topics since the first concepts were introduced, but this growth seems to be exponential nowadays, as it can be observed in Fig. 2.11, where the evolution of the amount of papers uploaded to Scopus webpage, the largest abstract and citation database of peer-reviewed literature, is presented. This means that there are lots of different approaches to study yet. Thanks to the computational and storage growth that were discussed in the previous section, everyone has access to AI courses where you can not only understand the first concepts about this technology but also to apply it to a new problem or try a new workaround without needing anything more computationally powerful than your own personal computer.

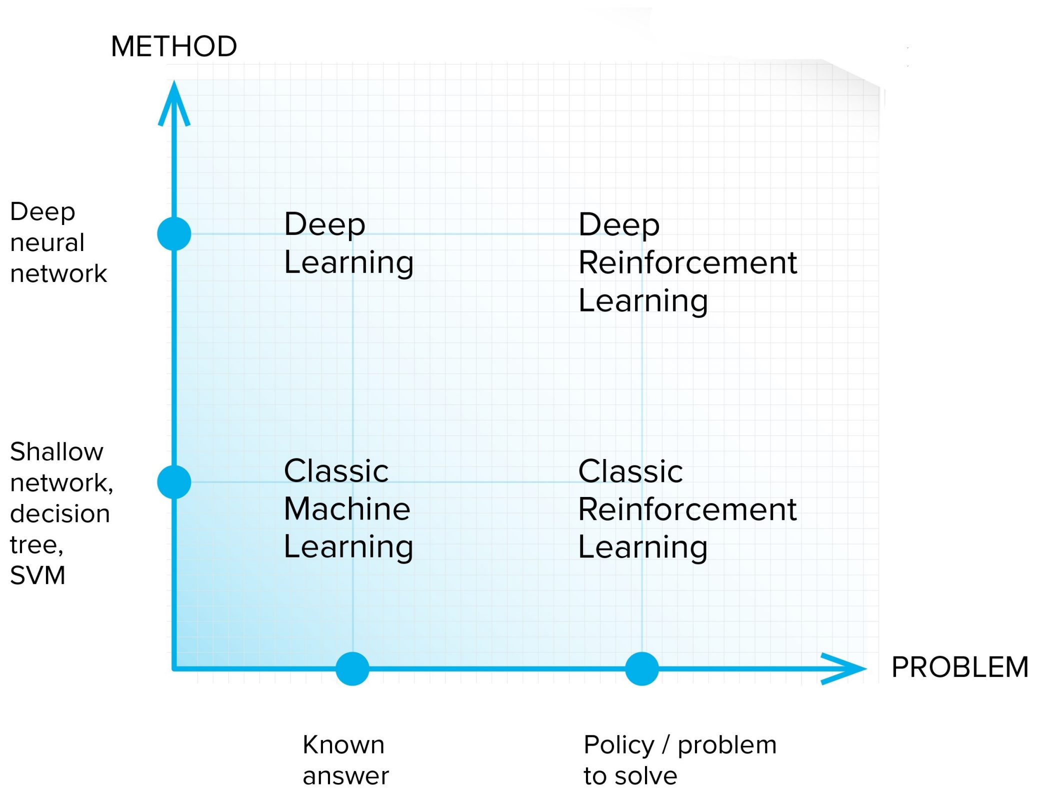
METHOD
PROBLEM
Deep neural network
Shallow network, decision tree, SVM
Known answer
Policy / problem to solve
Deep Learning
Deep Reinforcement Learning
Classic Machine Learning
Classic Reinforcement Learning

Figure 2.10 AI suitable techniques depending on the problem [25].

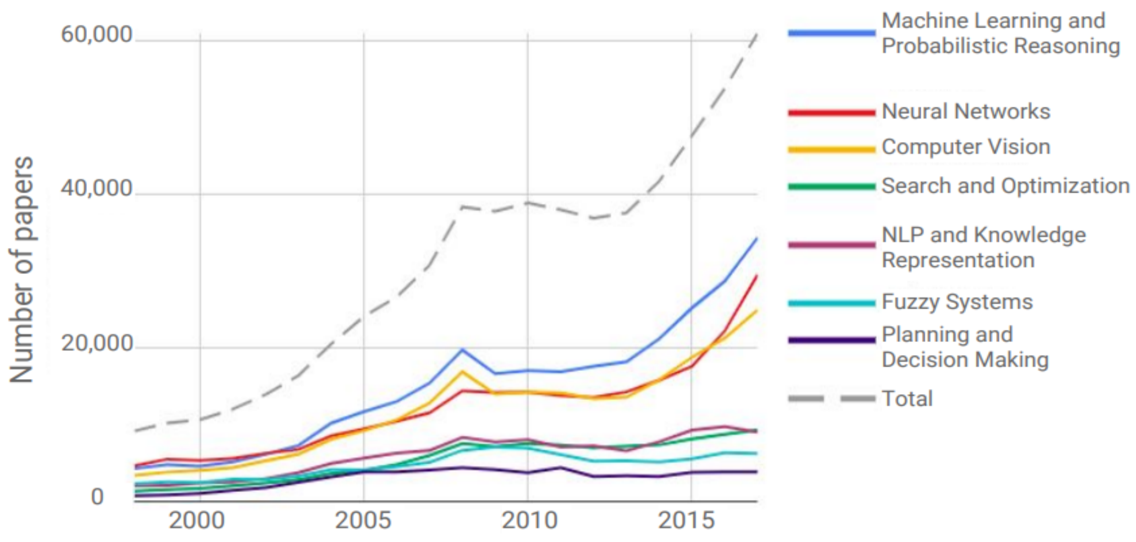

Figure 2.11 Number of papers in Scopus by subcategory [26].

The unceasing growth of deep learning research has also led to an increase its application into the business field. The foundation of AI startups is presented in Fig. 2.12. New applications and techniques are being found every day. The global business field is starting to take seriously into account this kind of algorithms when it comes to the matters of decision making or forecasting among other. This means that not only technological companies are getting benefit from this AI disrupt growth but also traditional companies such as banks or important clothing brands. Companies from very different fields are competing in the market in order to get the best specialists on these technologies, since this recent breakthrough has lead to a significant lack of professionals.

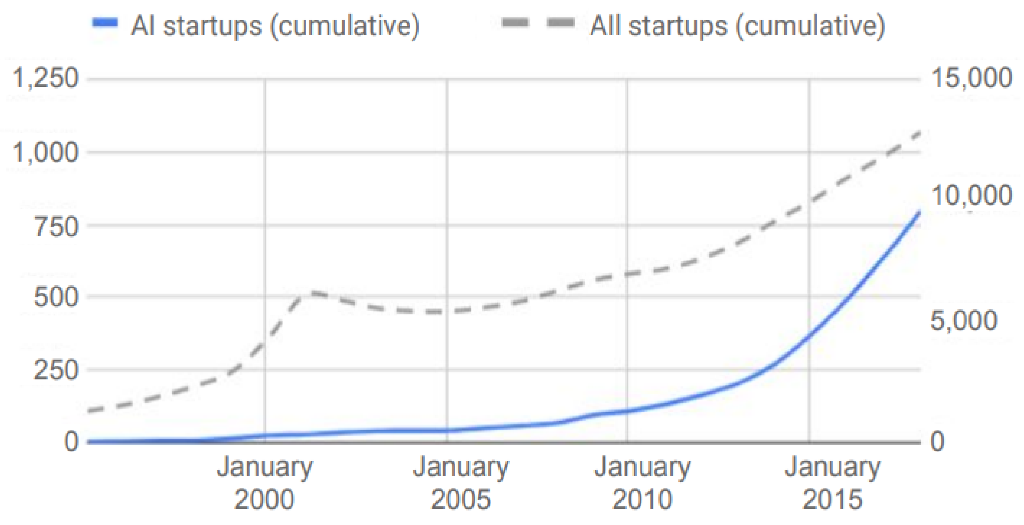

Figure 2.12 AI startups foundation growth compared to general startups growth [26].

All of these graphics clearly indicate that, although this technology has been increasing ever since it was created, it has recently experimented a disruptive growth. Whether this growth will continue or not, it depends on the success of the applications that are found every day and the ability of specialists about finding new areas to develop new AI concepts, techniques and applications.

3 Artificial Neural Networks

Ever tried. Ever failed. No matter. Try Again. Fail again. Fail better.

Samuel Beckett

Artificial Neural Networks (ANNs) have recently emerged as powerful tools in the scientific field. They are based on the usage of deep learning techniques and algorithms. This chapter will describe their structure and behavior in order to clear out the reasons why ANNs are becoming so helpful, specially on behavioral modeling [27].

3.1 Basic Structure

Understanding the concepts behind a neural network is not a trivial task. ANNs are often seen as black-box models, hence some people tend to misuse them just in order to get a problem solved without worrying about efficiency or computational power wastage. However, understanding the concepts inside this *black-box* could be helpful for fitting an ANN to a specific problem which definitely lead to better results in the end. The basic node of a neural network is called neuron (or unit). Each neuron receives an input and applies an activation function to compute the output.

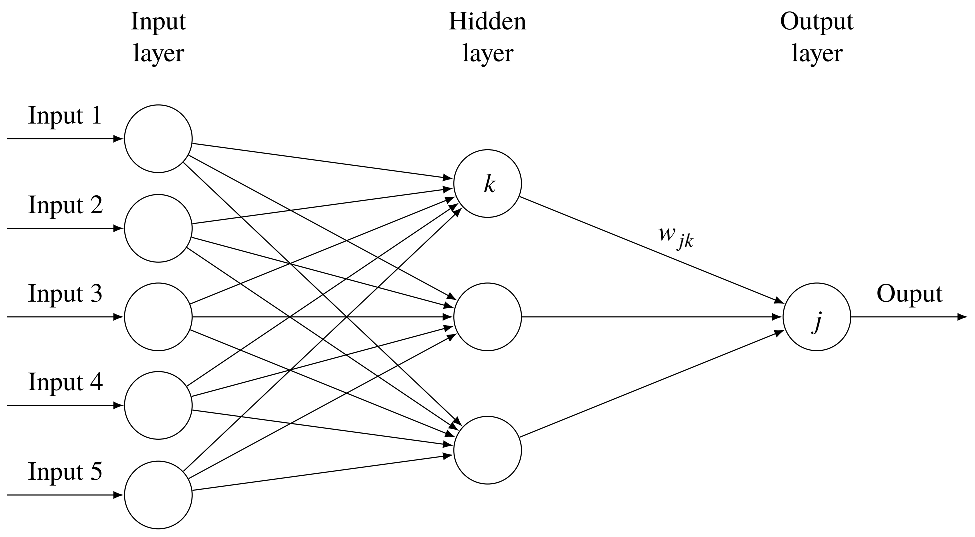

Figure 3.1 ANN basic structure and used notation.

Neurons are grouped together in layers. The first layer of an ANN is called input layer. This layer represents the connection between the network and the outside world. The number of neurons of the input layer must match the amount of inputs that are provided to the network. Behind the input layer, there will be a certain number of hidden layers, depending on the structure of our ANN. Neurons from consecutive layers are connected by weights, namely, the outputs of the different neurons in a layer are used to compute a weighted summation which will result as the input of the next layer neurons. The output y_j^l of the j-th neuron in layer l will be computed as

$$y_j^l = \Phi_j^l \left(\sum_k w_{jk}^l y_k^{l-1} + b_j^l \right), \tag{3.1}$$

where the superscript l stands for the position of the layer, w_{jk}^l is the weight between the k-th neuron of the layer $l-1$ and the j-th of layer l, b_l^j is the bias for the j-th neuron, which can be understood as an offset and $\Phi(\cdot)$ is the activation function. This process is called *forward propagation*. In order to simplify the mathematical description of this chapter, v is defined as

$$v_j^l = \sum_k w_{jk}^l y_k^{l-1} + b_j^l. \tag{3.2}$$

A graphical explanation about the mathematical operations that are used inside each neuron is presented in Fig. 3.2.

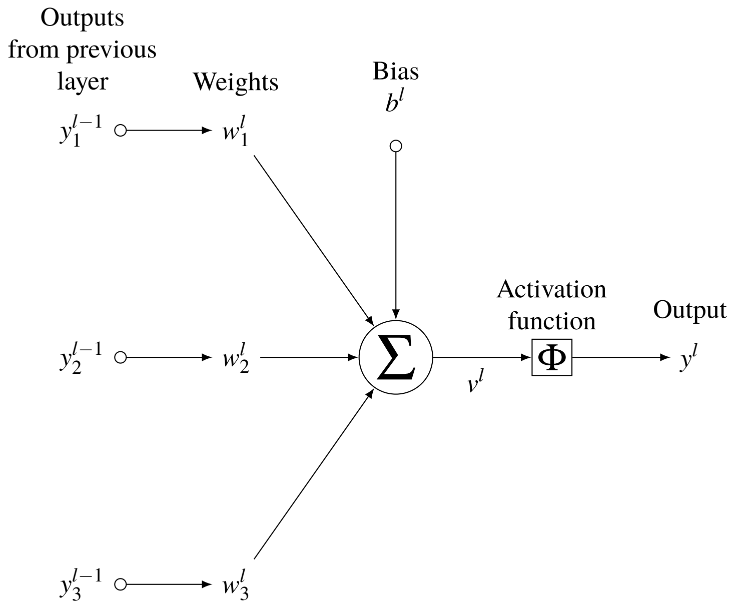

Figure 3.2 Computational model of a single neuron..

There may be as many hidden layers and neurons as necessary. However, the universal approximation theorem [28] states that single-hidden-layer ANNs, also known as TLPs (Three Layer Perceptrons), perform usually sufficient accuracy due to their universal approximation capabilities [29]. The general structure of a TLP is shown in Fig. 3.1. This structure provides ANNs the power to imitate any kind of nonlinear behavior as useful tools. They are often used as effective classifiers, regressors or even predictors.

3.2 Training

Adjusting a neural network to a specific situation requires a large amount of data. This data is divided in two datasets: test and training set. The latter is first provided as an input to the network in order to find the specific weights which reproduce the desired nonlinear behavior.

3.2.1 Forward Propagation

Defining the weights of each layer as a matrix following $\mathbf{w}^l = [w_{11} w_{21} w_{22}; w_{21} w_{31} w_{41}]$. During the training process, the ANN will receive inputs and it will compute the outputs following the expression

$$y^l = \Phi\left(\mathbf{y}^{l-1} \cdot \mathbf{w}^l + \mathbf{b}^l\right), \tag{3.3}$$

where $\mathbf{y}^{l-1}$ refers to the output matrix of the last hidden layer. The difference between the real and the computed output is calculated at each output neuron as a scalar cost

$$e_j = d_j - y_j, \tag{3.4}$$

where d_j refers to the desired output. We may now define the cost function as the mean squared error (MSE):

$$\varepsilon = \frac{1}{2}|\mathbf{e}|^2, \tag{3.5}$$

where $\mathbf{e}$ is a column vector containing the computed errors of each output neuron. This is where backpropagation algorithm takes place. This method computes the gradient of the cost function, which depends on weights. Afterwards, an optimization algorithm is used to minimize the cost according to the computed gradient. There are several variations of optimizers such as SGD, RMSProp or AdaGrad among others. This optimizer is used to identify the ANN suitable weights or, in other words, it enables the learning.

3.2.2 Backpropagation and Optimization

To understand how this learning is implemented through these two algorithms we first have to consider our initial situation. Considering that we want to create an ANN in order to make a regression of a specific function. Once the network architecture is defined, it can be trained. Before the training, the weights of each summation of inputs which are introduced to the different neurons are randomly initialized. Then, the aforementioned training starts. The ANN will take a few rows of inputs and compute the corresponding outputs for them through forward propagation. After that, the gradient of the cost function will be calculated. There are several different error functions that may be used for this purpose depending on our interests. Here, we will define the cost function as:

$$\varepsilon = \frac{1}{2}\sum_{i=1}^{N}(y_i - d_i)^2\,, \tag{3.6}$$

being N the total number of neurons in the final layer, in other words, the output dimension. At this point, we need to consider two possibilities. If we are computing the gradient at the output layer, we can do it instantly using the rule of chain

$$\frac{\partial \varepsilon}{\partial w_{jk}} = \frac{\partial \varepsilon}{\partial e_j} \cdot \frac{\partial \mathrm{e}_j}{\partial y_j} \cdot \frac{\partial y_j}{\partial v_j} \cdot \frac{\partial v_j}{\partial w_{jk}}. \tag{3.7}$$

From now on, during this section we will talk about a hidden neuron k and an output neuron j, which are connected through the weight w_{jk} as it is shown in Fig. 3.1.

However, if we are computing this gradient in a hidden layer, we will have to face what is named as the credit assignment problem [30]. In other words, we have to compute the amount of responsibility of each hidden neuron output for the cost function value. Using again the rule of chain, we find that

$$\frac{\partial \varepsilon}{\partial y_k} = \sum_j e_j \frac{\partial e_j}{\partial y_k}. \tag{3.8}$$

Now we have continue using the rule of chain, going backwards on our structure

$$\frac{\partial e_j}{\partial y_k} = \frac{\partial e_j}{\partial v_j}\frac{\partial v_j}{\partial y_k}, \tag{3.9}$$

where v_j is the result of the weighted summation of outputs from the hidden layer and y_k is the output of neuron k on the hidden layer. Differentiating Eq. 3.1 we obtain that

$$\frac{\partial v_j}{\partial y_k} = w_{jk}. \tag{3.10}$$

Hence, using Eq. (3.3) and

$$y_j = \Phi(v_j), \tag{3.11}$$

we compute we compute the error derivative as as

$$\frac{\partial e_j}{\partial v_j} = -\Phi_j^{'}(v_j) \tag{3.12}$$

Now, combining Eq. 3.12 and Eq. 3.8 it can be concluded that

$$\frac{\partial \varepsilon}{\partial y_k} = -\sum_j e_j \Phi_j^{'}(v_j) w_{jk} \tag{3.13}$$

Again, to simplify this equation and the following explanation we define the local gradient δ_j as

$$\delta_j = \frac{\partial \varepsilon_j}{\partial v_j} = \frac{\partial \varepsilon_j}{\partial e_j}\frac{\partial e_j}{\partial y_j}\frac{\partial y_j}{\partial v_j}, \tag{3.14}$$

so that when we reach the output layer the local gradient is easily computed as

$$\delta_j = -e_j \phi^{'}(v_j). \tag{3.15}$$

We may now rewrite Eq. 3.12 as

$$\frac{\partial \varepsilon}{\partial y_k} = -\sum_j \delta_j w_{jk}. \tag{3.16}$$

If we combine Eq. 3.15 and Eq. 3.13 have eventually found the formula that allows us to compute the local gradient at any neuron of our ANN, known as the backpropagation formula:

$$\delta_k = \phi^{'}(v_k) \sum_j \delta_j w_{jk}. \tag{3.17}$$

Once we have computed the local gradient using backpropagation algorithm, it is time for optimization. Our goal is to reduce the cost function by adapting the weights of each connection of neurons, having previously solved the credit assignment problem using Eq.3.12. In order to achieve this target we define the delta rule as

$$\Delta w_{jk} = -\eta \frac{\partial \varepsilon}{\partial w_{jk}}, \tag{3.18}$$

where η is a called the learning rate and the minus sign means that we want to go in the direction of gradient decrease.

As a conclusion, considering both of the cases that we talked about at the beginning of this section the update equation will be

$$\Delta w_{jk} = \eta \cdot e_j \cdot \Phi_j^{'}(v_j) \cdot y_k = -\eta \cdot \delta_j \cdot y_k, \tag{3.19}$$

if we are talking about the output layer and

$$\Delta w_{jk} = -\eta \cdot \Phi_j^{'}(v_j) \cdot \sum_j \delta_j w_{jk}, \tag{3.20}$$

otherwise. This adjustment enables the ANN to find specific correlations between the different inputs and the set of outputs. This process is repeated for N iterations, until the cost function converges to a certain value, which will depend on the problem and the ANN characteristics. Thanks to this algorithm, ANNs have been tested to be accurate at nonlinear systems modeling such as power amplifiers (PA) [31]. The number of rows that an ANN receives into the network before it applies backpropagation is called batch size and it can also be chosen by data scientists. The training ends when the whole training set has been introduced to the network N times where N is named number of epochs. Parameters whose values are chosen before the training starts, such as the number of neurons or hidden layers, the activation function, optimizer and batch

size among others are called hyperparameters.
There is not an exact methodology to find the best fitting characteristics of a certain model and they are often selected through success and error methods. Hence, experienced programmers on fitting neural networks into different models will likely find a better selection of hyperparameters than amateurs. This model tuning can be a nightmare for beginners. The process of optimizing the parameter values in a problem is called grid search. An accurate hyperparameter selection is essential for our model and it can determine whether the ANN will learn or not. Fortunately, there are implementations of this optimization process in order to facilitate grid search for amateurs, such as *GridSearchCV*.

GridSearchCV *GridSearchCV* is a python implementation of the parameter tuning optimization process, which is provided by the *sklearn* library [32]. It works with the model that we want to optimize and a list of the parameters that are going to be automatically tuned. Of course, some of these parameters are categorical variables, such as the functions or the optimizer algorithm. Therefore, they only have finite set of possible values. However, other parameters such as the number of layers or neurons in each of the layers have natural number values and it would make it computationally impossible to try every single possibility. That is why, besides the list of parameters to tune, another list is needed for each one of the parameters. This second list must contain the possible values of each hyperparamter that the programmer is interested to try and compare, since not every possible value can be tested. Eventually, GridSearchCV will create a model for each of the combinations of the possible values of each hyperparamter and it will evaluate the performance of these models in order to find the most accurate. Eventually, the most accurate combination of values and its model performance will be provided as output of the grid search process.

3.3 Validation

After the end of the training, it is time to evaluate our network depending on the ability of the ANN to deal with unseen data and compute its outputs, which is called generalization. Here, the test set is introduced to the ANN but the difference between computed and real outputs is not backpropagated to the network, that means that weights are not updated during this process. This difference will be used to calculate the accuracy of our ANN so that we can validate its behavior. As a matter of fact, validation is a very important stage since, depending in its results, the model will be considered adequate or not. Hence, is important to understand that obtaining a desired MSE during the testing stage may not be the end of our modeling task. Due to the random nature of the ANN, accuracies obtained with the same ANN of different test sets will never be the same. That is why not only a high accuracy is needed, but also a low variance. This is known as the bias-variance problem. K-fold cross validation [33] solves this issue by splitting the training set in k folds, being $k = 10$ very usual for a general application, so that there are k combinations of test and train sets which are introduced into the network in order to compute several accuracies. This technique allows us to calculate a reliable variance of the accuracy so that we can test our model on its generalization task.

k-Fold Cross-Validation

Accuracies computed in the test and train sets of a classic validation process may be very different between each other. This differences may be due to a bad modeling or hyperparameter tuning. However, differences between the test and train sets may also be a reason for that. That is why cross-validation emerged as a resampling procedure to evaluate machine learning models. It is a validation process that is not based on the test set train set division. It has become popular due to the fact that it has been proven to have lower bias than other methods and it is easy to implement. This approach is based on dividing data into folds and ensuring that each fold is tested once. After randomly shuffling the dataset and dividing it in k subsets, an iterative process starts. Each one of the iterations of this iterative process follow this curse :

1. Select a fold or subset as the test set of that iteration.
2. Select the rest of them as training set.
3. Fit the model using the $k-1$ training sets together.
4. Evaluate the model using the selected test set of that iteration.
5. Take note of the evaluated performance on the model and start the next iteration.

After this loop, a list of evaluation scores will have been computed. Therefore, we only need to compute the mean of this error values in order to obtain a better evaluation of our model. It is important to understand the fact that each data sample is assigned to a subset in the fist division of the randomized dataset an this assignment must not change during the whole process. Hence, each sample is given to the model $k-1$ times as a training sample and 1 time as test sample. As it can be inferred from its iterative nature, *k-Fold Cross-validation* takes longer than the train-test sets division validation. An illustration of the process is presented in Fig. 3.3.

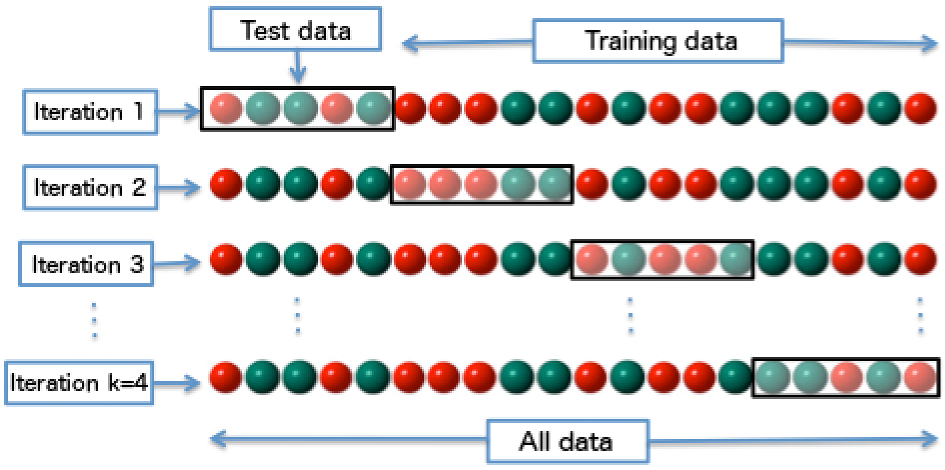

Figure 3.3 *k-Fold Cross-validation* graphical process [34].

3.4 Improving our ANN

Overfitting and Underfitting After calculating the accuracy of our network we may encounter a substantial problem. If our computed MSE differs a lot from the MSE obtained during the training problem, we are probably dealing with *overfitting*. This happens because our ANN has found certain specific correlations on the training set that are unique of that data set. Therefore, our ANN fails at the generalization. Those unique correlations may have been found due to excessive training on the set, maybe by using too many neurons or epochs. A helpful workaround for this issue is *dropout*. This is a simple technique to prevent overfitting. It consists on leaving a determined percentage of neurons asleep during each iteration of the training process. Of course, the ignored units are be randomly chosen at each iteration. Underfitting, on the

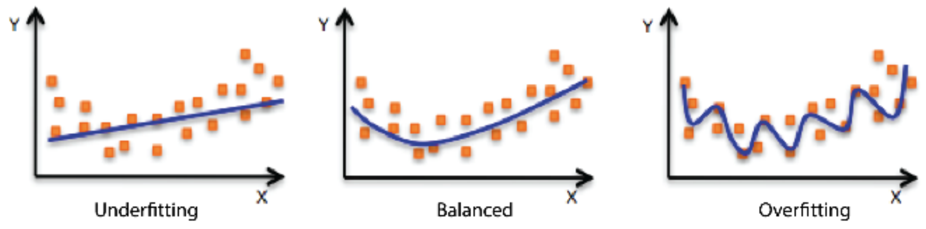

Figure 3.4 Underfittted, balanced and overfitted regression [35].

contrary, happens when the model is not complex enough or it has not trained as much as it takes to identify the necessary input-output correlations in order to get accurate results. Overfitting is a more popular problem than underfitting, but both of them need to be taken into account when results are not as they were expected. Identifying this kind of problems is essential to avoid time wastage on tuning hyperparameters without having a clue of what is really going on, that is why Fig. 3.5 and Fig. 3.4 are provided as graphical explanations.

Vanishing and Exploding Gradient Another problem that may get in the way of making our ANN learn is named vanishing gradient. It happens when, during the training start, the gradient of the loss function at the layers comes closer and closer to zero, thus the network stops learning or does it very slowly. The opposite problem, which is named exploding gradient, may also occur. Exploding gradient appear when error

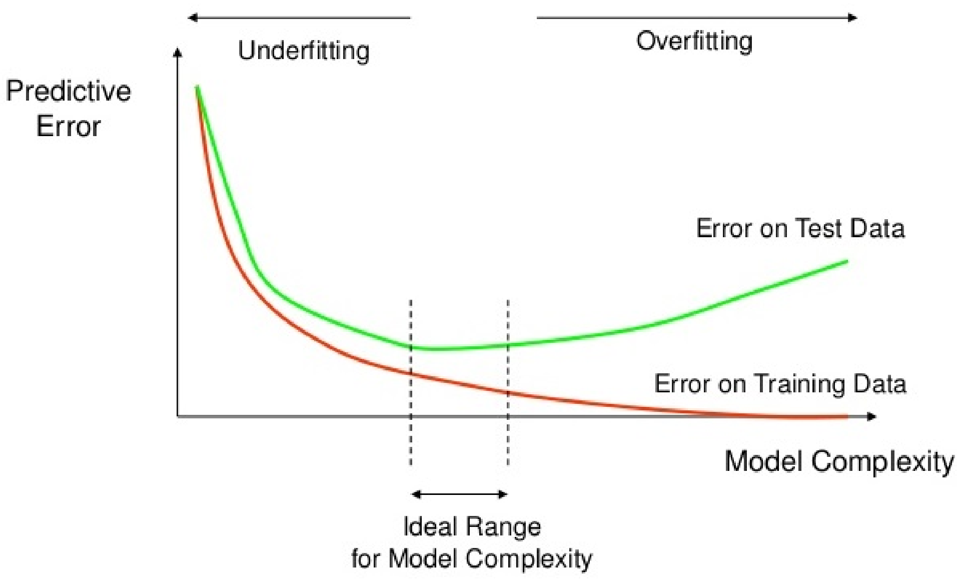

Figure 3.5 Model complexity relevance on test and train error [36].

gradients become larger at every backpropagation step, provoking huge unwanted updates to neural network model weights during training. Both of this problems make the model unable to perform successfully the gradient descent algorithm in order to find the minimun of the cost function. The reason why it happens is related with the backpropagation algorithm. As, during backpropagation, the chain of rule is used to calculate this derivatives in order to solve the *credit assignment problem*, the derivatives of the activation functions of different layers are multiplied between each other. When some of them tend to values lower than one, it causes a substantial decrease of the derivative so that its value make the weights to change very slowly. This happens when using activation functions that have low derivative values when low or high inputs are introduced to them, such as the sigmoid activation function. Despite being one of the most popular ones thanks to its other properties, it sometimes triggers this issue due to its small derivative. Using other activation functions such as ReLu solves effectively this issue. Another workaround that is worth mentioning is called batch-normalization layers. This type of layers are used to normalize the inputs of the original layers of the neural network in order to avoid the inputs from reaching extreme values, so that the gradient of the activation function does not decrease dramatically when applying backpropagation. Residual neural networks have also been presented as an alternative. They present some straight connections between layers, without application of any activation functions. This straight connection is added to the output of the layer in order to avoid the vanishing gradient.

3.4.1 Preparing the Input Dataset

Another issue that needs to be studied before training our ANN is preparing our input dataset. Collected dataset are not usually ready for being used as input to an ANN. ANNs input datasets need to accomplish several requirements in order to allow the model to get the best of it and learn as much as it can. Firstly, categorical variables need to be encoded. Until now we have been working under the presumption that variables are just numbers but lots of datasets are based on categories and numbers at the same time. Transforming this names or classes into numbers will be our first task about making the dataset suitable for our ANN. This process is call enconding and there are two different approaches which need to be discussed within this section. Let's suppose the case in which we are dealing with some human resources data base. In this database we have information about people and one of the fields refers to each person nationality. Now, as an ANN cannot work with text, we need to transform nationalities to numbers. Label Enconding is one of the possible workarounds. This solution will identify each nationality with an integer number and substitute it in the input dataset. Therefore, if we had input nationality column with values like {Spanish, German, French...} it will be converted to {1,2,3...}. This is a simple and computationally efficient solution. However, it ANN will receive this input column and treat is as it treats others in order to get some information from it. Although these numbers are just being used as symbols, it is possible that our model finds some correlation or hierarchy between them and. This problem would surely decrease the model accuracy. That is why One

Hot Enconding is presented. This workaround is based in the creation of a new column for each categorical value in the input dataset. The method is feasible for categorical variables with low amount of possible categorical values since, otherwise, the number of columns starts increases excessively. In each one of the new columns, every row will be set to zero except of the one that actually belongs to that category. According to our previous example, each nationality would be a new column so that Spanish nationality column would be like $\{1,0,0...\}$, which indicates that the first person is Spanish and the others are not. This avoids the issue of our ANN finding correlations between rows of categorical columns but it does it with a computational cost which can be impossible to manage at some cases. Now that the input dataset is completely filled with numeric data, it may be helpful to normalize it. Although it is not mandatory to normalize our input, the truth is that ANNs have been proven to be more efficient when dealing with normalized input datasets. Gaussian normalization is computed as $x_{norm} = \frac{x-\mu}{\sigma}$, where μ represents the mean of the input and σ the standard deviation. It is actually the most popular normalization although depending on the problem one kind or another will be more suitable. As an example, min-max feature scaling is also widespread and is computed as

$$norm = \frac{x - min(x)}{max(x) - min(x)}. \tag{3.21}$$

Activation Functions

Finally, it is worth mentioning the importance of hyperparameters. As it has already been explained, this concept includes the different parameters of a neural network that must have a fixed value before the training start. This value must be chosen carefully by the engineer, considering the application and nature of the problem itself. Otherwise, a simple problem may be impossible to solve by an ANN with wrong hyperparameter values. As a matter of fact, that is why some people may wrongfully believe that an ANN is not suitable for the problem they are facing just because they tried a general implementation, without any hyperparameter adjusting. Among all the hyperparameters, activation functions are critically decisive to the results. As it has already been explained, activation functions compute the output of a certain input at each neuron. Somehow, activation functions determine how a neuron reacts to the different inputs. They are usually defined layer to layer, meaning that all of the neurons in one layer generally use the same activation function, although this is not mandatory. Activation functions are basic to enable our ANN to learn a nonlinear behavior. Through the different activation functions of each layer, a complicated nonlinear behavior which would never be reproduced by just one layer may be divided in nonlinearities that can be reproduced at each one of the layers. That is why, as matter of fact, ANNs are actually considered as universal approximators [37]. Computational efficiency is required not only to the activation function of a model but also to its derivative, since it will be computed lots of times during a training process. Among all of the different activation functions [38], it has been considered important to highlight the most popular ones and their properties in this section:

- **Step function:** It is a threshold-based function. It only have two possible values $\{0,1\}$ which makes it interesting for classification purposes, since it can normalize the output. This means that, if the input is above zero, the neuron just let this input pass to the next layer without being affected. Otherwise, if its is negative or zero valued, the neuron do not let the input pass to the next layer, so its output remains null. It may be seen as if the neuron gets *fired* or not depending on the input.

$$f(x) = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases} \tag{3.22}$$

- **Sigmoid function:** It is the most popular one. It is helpful to compute percentages and probabilities, since its values are limited from zero to one $[0,1]$, so it is often used as an enhancement of the step function when classifying binary variables. Its formula is

$$f(x) = \frac{1}{1+e^{-x}} f'(x) = \frac{e^{-x}}{(1+e^{-x})^2}. \tag{3.23}$$

As it can be observed in the equation, its derivative is a smooth function. This avoids abrupt changes on its derivative value, so it works better for backpropagation. Its major drawbacks are involved with its computational cost and the fact that it may cause vanishing gradient. It is not zero centered, which may be a problem in some cases.

- **Hyperbolic tangent function:** It is similar to the sigmoid function but it is zero centered, which is interesting to model inputs with negative, positive and neutral values. This is its equation

$$f(x) = \tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}}. \tag{3.24}$$

- **Linear function:** It is the most simple one. It always let the input pass through the neuron without being affected. Although it may seem useless for the described purpose of this section, it is actually very popular since it is very helpful at the output layer. As the reader can imagine, it does not help to approximate nonlinearities but, at the output layer, it is used just to compute the weighted summation from the output of the previous layer neurons. Its major drawback is related with the fact that, since its derivative value is always one, it does not help backpropagation algorithm at all.

- **ReLu function:** It solves the backpropagation difficulty presented by the linear activation function, since its derivative is the step function. However, it does not work well when input are mostly negative due to the fact that its derivative is null in those cases and therefore backpropagation is cut off. Its formula is

$$f(x) = \begin{cases} 0 & x \leq 0 \\ x & x > 0 \end{cases} \tag{3.25}$$

- **Leaky ReLu function:** It is presented as an evolution of the ReLu function, which solves the problem of the null derivative with negative inputs, also known as the dying ReLu problem. Its behavior is mathematically expressed as

$$f(x) = \begin{cases} \alpha x & x \leq 0 \\ x & x > 0 \end{cases}, \tag{3.26}$$

  where α is a positive number smaller than 1. This function introduced the concept of parametric ReLu functions, a set of functions that try to improve the ReLu function using parameters and preventing, through different ways, the dying ReLu problem.

A graph has been included in Fig. 3.6 in order to illustrate a graphical representation of the presented activation functions are their mathematical behavior.

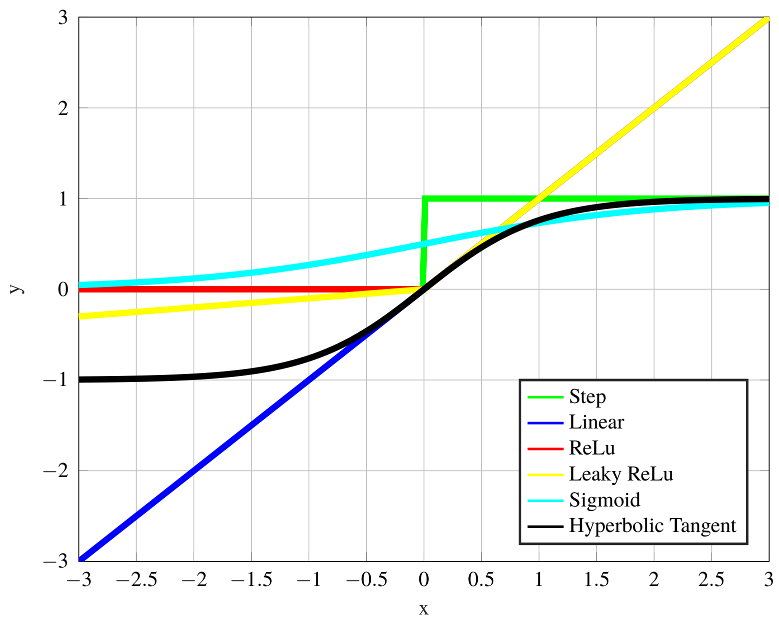

Figure 3.6 Most popular activation functions.

3.5 Complex Valued Neural Networks

Complex Valued Neural Networks (CVNNs) are based on the usage of complex numbers with the structure of an ANN [39]. CVNNs have been tested to perform better accuracy results for complex input and output datasets, when the accurate structure is found [40]. Since weights, biases and activation functions are complex numbers, the LMS algorithm cannot be used to reduce the cost function value. Complex Least Mean Square (CLMS) [41] algorithm makes use Wirtinger derivatives to solve this issue on a CVNN [42]. Several approaches such as [43] and [44] have been studied during the past years to work with complex variables at ANNs.

3.5.1 Complex Least Mean Square Algorithm

Considering the fact that the output is now complex, the error, computed as shown in Eq. (3.4), is also complex and that allows the redefinition of the cost function introduced in Eq. (3.5) as

$$\varepsilon = \frac{1}{2} e \cdot e^{*}. \tag{3.27}$$

Having already redefined the cost function, the next step is to compute the update equation for complex weights defined as

$$\Delta \mathbf{w} = -\eta \frac{\partial \varepsilon}{\partial w^{*}}, \tag{3.28}$$

where η is always a positive number that modulates the step. Making use of the Wirtinger derivative, $\frac{\partial \varepsilon}{\partial \mathbf{w}^{*}}$ is computed as

$$\frac{\partial \varepsilon}{\partial \mathbf{w}^{*}} = \frac{1}{2} \frac{\partial e \cdot e^{*}}{\partial w^{*}} = -\frac{1}{2} e^{*} \frac{\partial y}{\partial w^{*}} - \frac{1}{2} e \frac{\partial y^{*}}{\partial w^{*}}. \tag{3.29}$$

Considering the usage of

$$y = v \cdot |v|^{n-1} \tag{3.30}$$

as activation functions, where n is $n = 2i + 1$ being i a non-negative intege which indicates the position of the neuron in a layer, the partial derivatives of the output y and its complex y^{*} and of a neuron with respect of w^{*} are calculated as

$$\frac{\partial y}{\partial w^{*}} = \frac{\partial y}{\partial v} \frac{\partial v}{\partial w^{*}} + \frac{\partial y}{\partial v^{*}} \frac{\partial v^{*}}{\partial w^{*}}, \tag{3.31}$$

and

$$\frac{\partial y^{*}}{\partial w^{*}} = \frac{\partial y^{*}}{\partial v} \frac{\partial v}{\partial w^{*}} + \frac{\partial y^{*}}{\partial v^{*}} \frac{\partial v^{*}}{\partial w^{*}}, \tag{3.32}$$

where $\frac{\partial v}{\partial w^{*}} = 0$ due to the lack of dependence between v and w^{*}. Now, using equation Eq. (3.2), it can be computed that

$$\frac{\partial v^{*}}{\partial w^{*}} = y^{*l-1}, \tag{3.33}$$

and considering Eq. 4.12, the partial derivatives $\frac{\partial y}{\partial v^{*}}$ and $\frac{\partial y^{*}}{\partial v^{*}}$ are

$$\frac{\partial y^{*}}{\partial v^{*}} = \frac{\partial y}{\partial v} = \frac{n+1}{2} |v|^{n-1}, \tag{3.34}$$

and

$$\frac{\partial y}{\partial v^{*}} = \frac{\partial y^{*}}{\partial v} = \frac{n-1}{2} \cdot v^{2} |v|^{n-3}. \tag{3.35}$$

Using these equations, the gradient of the complex cost function is computed as

$$\frac{\partial \varepsilon}{\partial w^{*}} = -\frac{1}{2} \mathrm{e} \frac{n+1}{2} |v|^{n-1} \cdot y^{*l-1} - \frac{1}{2} e^{*} \frac{n-1}{2} v^{2} |v|^{n-3} y^{*l-1}. \tag{3.36}$$

Therefore, the update equation for complex weights is

$$w = w + \frac{1}{2} \eta \left(e \frac{n+1}{2} \cdot |v|^{n-1} + e^{*} \frac{n-1}{2} v^{2} |v|^{n-3} \right) y^{*l-1}. \tag{3.37}$$

However, as we have already consider in a real-valued artificial neural network, we cannot use this update equation for hidden layers. Therefore, the credit assignment problem is required to be solved again [45]. In the case under consideration, Eq.(3.8) is now rewritten as

$$\frac{\partial \varepsilon}{\partial y_k} = \frac{\partial \varepsilon}{\partial e_j}\frac{\partial e_j}{\partial y_k} + \frac{\partial \varepsilon}{\partial e_j{}^*}\frac{\partial e_j{}^*}{\partial y_k} = \frac{1}{2}\left(\sum_j e_j{}^*\frac{\partial e_j}{\partial y_k} + \sum_j e_j\frac{\partial e_j{}^*}{\partial y_k}\right), \tag{3.38}$$

where j refers to an output node and k is used to index a hidden neuron [46]. In order to continue the mathematical elaboration, the partial derivatives of the output neurons error depending in hidden node output are calculated as

$$\frac{\partial e_j}{\partial y_k} = \frac{\partial e_j}{\partial y_j}\frac{\partial y_j}{\partial y_k} + \frac{\partial e_j}{\partial y_j{}^*}\frac{\partial y_j{}^*}{\partial y_k} = -\Phi_j'(v_j)w_{jk}, \tag{3.39}$$

having considered the fact that the order term of the Wirtiner derivative is null due to the lack of dependency between the error e_j of an output node and its conjugated output $y_j{}^*$. The last term of the equality in the previous equation has been computed considering that

$$\frac{\partial y_j}{\partial y_k} = \frac{\partial y_j}{\partial v_j}\frac{\partial v_j}{\partial y_k} + \frac{\partial y_j}{\partial v_j{}^*}\frac{\partial v_j{}^*}{\partial y_k} = -\Phi_j'(v_j)w_{jk}, \tag{3.40}$$

where the second term $\frac{\partial y_j}{\partial v_j{}^*}\frac{\partial v_j{}^*}{\partial y_k}$ is null again. Using the same logic, it is easy to compute the second term of Eq.(4.12) as

$$\frac{\partial e_j{}^*}{\partial y_k} = -(\Phi_j{}^*)'(v_j)w_{jk}. \tag{3.41}$$

We may now insert equations Eq.(4.13) and Eq.(4.14) in Eq.(4.12) to obtain a complete equation to compute the partial derivative of the error function depending on the output of a hidden neuron as

$$\frac{\partial \varepsilon}{\partial y_k} = -\frac{1}{2}\left(\sum_j e_j{}^*\Phi_j'(v_j)w_{jk} + \sum_j e_j(\Phi_j{}^*(v_j))'w_{jk}\right). \tag{3.42}$$

This equation allows us to compute the partial derivative of the error function depending on the output of a node which is just behind the output layer. However, if we wanted to compute it in a previous layer to this one, we would have to continue the mathematical development backwards. For this reason, it is interesting to redefine the local gradient. Starting from Eq.(4.11), the complex update is

$$\Delta\mathbf{w} = -\eta\frac{\partial \varepsilon}{\partial w^*} = \eta\left(\frac{1}{2}e_j{}^*(\Phi_j{}^*(v_j))' + \frac{1}{2}e_j\Phi_j'(v_j)\right)y^{*l-1}, \tag{3.43}$$

the local gradient of an output node may be easily identified as

$$\delta_j = \frac{1}{2}e_j{}^*(\Phi_j{}^*(v_j))' + \frac{1}{2}e_j\Phi_j'(v_j). \tag{3.44}$$

Finally, the local gradient of a hidden unit δ_k needs to be computed in order to implement a complete and scalable complex backpropagation algorithm. After redefining δ_k as

$$\delta_k = -\frac{\partial \varepsilon}{\partial y_k}\frac{\partial y_k}{\partial v_k} - \frac{\partial \varepsilon}{\partial y_k{}^*}\frac{\partial y_k{}^*}{\partial v_k} = -\frac{\partial \varepsilon}{\partial y_k}\Phi_k'(v_k) - \frac{\partial \varepsilon}{\partial y_k{}^*}(\Phi_k{}^*(v_k))', \tag{3.45}$$

replacing $\frac{\partial \varepsilon}{\partial y_k}$ Eq.(4.15) and doing a similar proceeding to compute

$$\frac{\partial \varepsilon}{\partial y_k{}^*} = -\frac{1}{2}\left(\sum_j e_j{}^*(\Phi_j{}^*(v_j))'w_{jk}^* + \sum_j e_j\Phi_j'(v_j)w_{jk}^*\right), \tag{3.46}$$

we can eventually express the local gradient of a hidden unit as

$$\begin{aligned}\delta_k =& \frac{1}{2}\left(\sum_j e_j{}^*\Phi_j'(v_j)w_{jk} + \sum_j e_j(\Phi_j{}^*(v_j))'w_{jk}\right)\Phi_k'(v_k)\\ &+\left(\sum_j e_j{}^*(\Phi_j{}^*(v_j))'w_{jk}^* + \sum_j e_j\Phi_j'(v_j)w_{jk}^*\right)(\Phi_k{}^*(v_k))'.\end{aligned} \tag{3.47}$$

Using this set of equations, every weight in the network can be updated in order to make our CVNN learn during the training stage.

4 Power Amplifiers and Behavioral Modeling

Everyone is a genius. But if you judge a fish by its ability to climb a tree, it will live its whole life believing that it is stupid.

Albert Einstein

Power amplifiers (PAs) represent a key role in wireless communications systems thanks to their ability to make up for the free-space path loss, which happens between the transmitter and the receiver. That is why they are used to achieve enough transmission power, specially in mobile applications. An accurate description about PAs, which is presented in [47], states: *"These amplifiers take a small input and make it stronger and larger creating a wider area of use with a more robust signal."* However, their behavior presents a conflict of interests between efficiency and linearity. This chapter will introduce the problem of power amplifiers modeling, its necessity and the different workarounds that have been studied to solve it [27].

4.1 Ideal Amplifier

An interesting approach to the behaviour of PAs in a wireless communications system involves starting from considering an ideal amplifier. Accordingly, before talking about non-desired effects found in real power amplifiers and how to manage them, it is important to enumerate the requirements of an ideal power amplifier. First of all, gain must be constant and independent in relation to the input signal power or frequency. This behavior is known as linear since that is how the input-output graph looks like. Mathematically speaking, a linear system is the one that satisfy the property of homogeneity and additivity or superposition [48]. A system behavior is described as homogeneous when an amplitude increase or decreasement in the input signal provokes a directly proportional amplitude change of the output. If we considered a system where $y = f(x)$, an homogeneous system would behave as

$$f(kx) = kf(x) = ky. \tag{4.1}$$

This requirement is also known as the scaling principle. Stated like this, it may be thought that every amplifier in the world satisfies this principle since, speaking in terms of voltage, an amplifier behavior is expressed as

$$V_{out}(t) = GV_{in}(t). \tag{4.2}$$

However, power dissipation and the energy conservation law will make real amplifiers fail at satisfying these requirements, so non-linearities are actually the general rule [49]. Following this notation, a system that accomplishes the principle of superposition behaves in a manner that an addition of its inputs results in an addition of the individual outputs of each of the inputs, without interacting between each other. This ability is mathematically expressed as

$$f(x_1 + x_2) = f(x_1) + f(x_2) \tag{4.3}$$

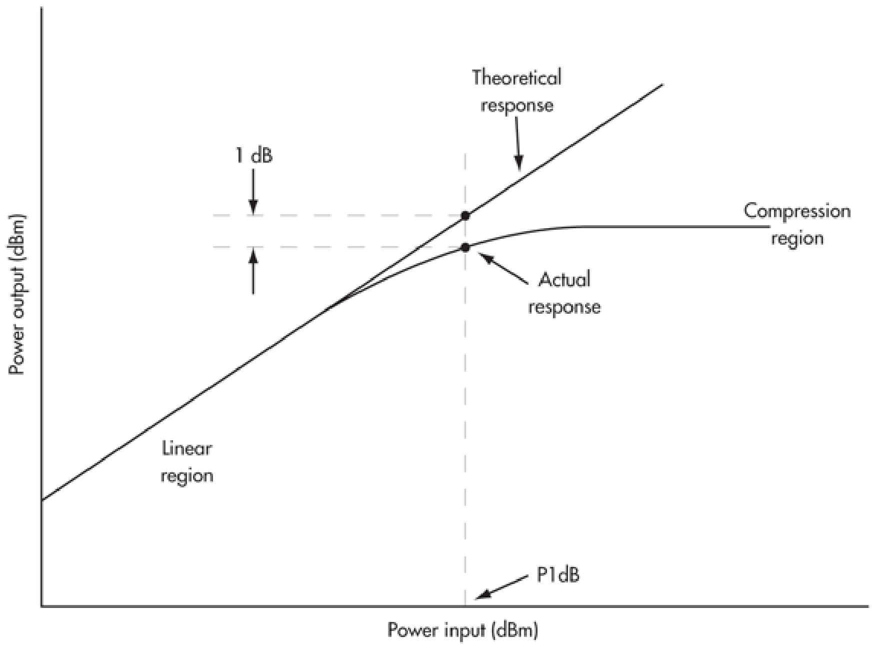

Figure 4.1 Ideal and real amplifier behavior [51].

4.2 Conflict of interests

4.2.1 Non-linear effects

Linearity is generally accepted as a required characteristic of an ideal amplifier. A linear behavior is easy to understand. We may think, therefore, that it is easy for an amplifier to behave linearly and that most of the amplifiers are linear. In this case, it would be a huge mistake, since reality is way more complex. The drawbacks of non-linear effects have not been discussed yet. Although the gain of an ideal amplifier is constant, real amplifier reach to a *compression zone* when a enough input power is provided. In this zone, the PA is saturated or, in other words, the gain of the amplifier stops being constant, since the output stops increasing no matter how much the input grows [50]. This is not a difficult effect to understand. Indeed, it happens because of the energy conservation law, which stands that the output power cannot be higher than the supply power. As a matter of fact, the output power will be lower than the supply due to the appearance of power dissipation as heat. Considering the gain of an amplifier as $G = \frac{P_{out}}{P_{in}}$ and making use of the energy conservation law, the gain of a real amplifier can be rewritten as

$$G = \frac{P_{in} + P_{DC} - P_{diss}}{P_{in}}, \tag{4.4}$$

where P_{diss} stands for the power wastage and the balance of power in this case allows us to rewrite the output power as $P_{out} = P_{in} + P_{DC} - P_{diss}$. If we substitute P_{out} for GP_{in} in (4.22), we can isolate G as

$$G = 1 + \frac{P_{DC} - P_{diss}}{P_{in}}. \tag{4.5}$$

From this equation we can conclude that the gain of a PA cannot be constant when the input power increases. Moreover, it can also be concluded that every element of a system that requires a supply voltage to work will inherently be non-linear. This happens because the supply voltage will be limited but the input power could change, considering the fact that the dissipated power value may be overlooked. Therefore, since the output signal may never exceed the supply voltage, it will be *clipped* when enough P_{in} is provided. The point of the input-output graph of a PA where the gain starts being $1dB$ lower than it ideally should be is named 1dB compression point. This critical point is illustrated in Fig.4.1.

This clipping effect in the time domain provokes the occurrence of harmonic distortion in frequency domain. The distortion of an electrical signal is defined as any change of the desired output wave, except for scaling or constant time delay. Harmonic distortion, particularly, is defined as the appearance of new output signals named harmonics, whose frequency are multiples of the fundamental frequency of the input signal. This leads to the growth of the spectral bandwidth. Total Harmonic Distortion (THD) percentage is an important technique to measure the level of harmonics in a voltage or current waveform [52]. It computes the relation between the total output power generated by harmonics and the power of the fundamental tone as it is shown in (4.6).

$$THD = \frac{\sum_{i>1} P_i}{P_1} = \frac{P_{total} - P_1}{P_1}, \tag{4.6}$$

where P_1 refers to the fundamental tone output power. Bandwidth growth is a non desired effect, since other signals could be affected by our harmonics and vice versa. However, it is not the most problematic issue. The reason is that, using a suitable filter, the tones which are not in the same band as the desired output can be removed. Unfortunately, some of those so-called harmonics appear with the same frequency as the input signal, the fundamental tone, so that the problem cannot be solved with filtering. This kind of distortion is called in-band distortion. In order to understand how harmonics appear, we will use a mathematical explanation. From now on, the considered input will be a general input signal in a telecommunications system such as a sinusoidal curve expressed like

$$x(t) = A(t)\cos(wt + \theta(t)), \tag{4.7}$$

where t refers to time and the frequency $w = 2\pi f = \frac{2\pi}{T}$, being T the period of the sinusoidal. When an input signal like the one described in (4.22) is introduced to the PA of a wireless communication systm, the information resides in the amplitude (AM) or the phase (PM) variation through time. A linear system would compute an output signal as following

$$y_L(t) = \alpha A(t)\cos(w(t-\tau) + \theta(t-\tau)), \tag{4.8}$$

where α is a constant that can be higher or lower than one, in other words, it can amplify or reduce the amplitude of the output, and τ is used to represent the memory effects as a possible delay. We can observe that in a linear system, the output signal only differs with the input by a scaling constant and a positive finite time delay. Adding higher order dependencies is a direct way to reproduce a non-linear behavior of a system. Therefore, a non-linear system could be expressed, if we only considered second order non-linearities as

$$y_{NL}(t) = \alpha xt - \tau) + \beta x^2(t-\tau), \tag{4.9}$$

which, if we use as input our previously described sinusoidal curve, can be rewritten as

$$y_{NL}(t) = \alpha A(t)\cos(w(t-\tau) + \theta(t-\tau)) + \beta A^2(t)\cos^2(w(t-\tau) + \theta(t-\tau)). \tag{4.10}$$

Considering the trigonometric relation between the double-angle cosine and $\cos^2(\alpha)$ as

$$\cos^2(\alpha) = \frac{1+\cos(2\alpha)}{2}, \tag{4.11}$$

the non-linear output of a telecommunication system can be expressed as

$$\begin{aligned} y_{NL}(t) &= \alpha A(t)\cos(w(t-\tau) + \theta(t-\tau)) + \beta A^2(t)\frac{1+\cos(2w(t-\tau) + 2\theta(t-\tau))}{2} \\ &= \frac{\beta A^2(t)}{2} + \alpha A(t)\cos(w(t-\tau) + \frac{\beta A^2(t)}{2}\cos(2w(t-\tau) + 2\theta(t-\tau)). \end{aligned} \tag{4.12}$$

In (4.12) it can be observed that a second order non-linearity adds what is known as DC voltage term to the output power, this type of nonlinear behavior is known as self-biasing. In addition to this effect a new sinusoidal signal named harmonic with twice the frequency of the input signal is added to the output. This reasoning describes the emergence of harmonics but a second order non-linearity does not affect to the amplitude of the fundamental tone, thus it can be solved by filtering. However, if we did the same process for third order non-linearities following the expression in (4.10), the new $y_{NL}(t)$ can be expressed now as

$$y_{NL}(t) = \alpha x(t-\tau) + \beta x^2(t-\tau) + \gamma x^3(t-\tau). \tag{4.13}$$

Using other trigonometric identities such as

$$cos^3(\alpha) = \frac{3cos(\alpha)}{4} + \frac{cos(3\alpha)}{4}, \tag{4.14}$$

the output of a third order nonlinear system can be rewritten as

$$\begin{aligned} y_{NL}(t) = \frac{\beta A^2(t)}{2} + \alpha A(t)\cos(w(t-\tau) + \gamma\frac{3}{4}A^3(t)\cos(w(t-\tau)+\theta(t-\tau)) + \\ \beta A^2\frac{1}{2}\cos(2w(t-\tau)+2\theta(t-\tau)))\frac{1}{4}\cos(3w(t-\tau)+3\theta(t-\tau)). \end{aligned} \tag{4.15}$$

With the addition of a third order non-linear term, instead of a new DC component, what we get is an amplitude distortion of the fundamental tone whose amplitude is now $\alpha A(t) + \gamma\frac{3}{4}A^3(t)$. This in-band distortion provokes changes in the output signal that cannot be solved by the usage of filtering. If we continued using higher order nonlinearities to represent our PA behavior, we would conclude that the output signal can be expressed as

$$y_{NL}(t) = y_0 + \sum_{k>1}^{N} y_k cos(w_k t + \phi_k), \tag{4.16}$$

being $w_k = 2\pi fk$. If we observe in detail both of the previous mathematical elaborations, we will realize the fact there is a significant difference between even and odd order nonlinearities. The latter affect directly to the fundamental tone by changing its amplitude, while the first ones are not responsible for in-band distortion and can be solved with filtering.

Power amplifiers are formed of different elements. Among them, there are energy storing components. This energy storage provokes memory effects in our output signal, meaning that the output does not simply depend on the input at a certain moment, but also on the previous inputs, as shown in (4.23). In addition to amplitude distortion, a nonlinear system may include phase distortion, meaning that it does not respond equally to frequency changes, in terms of memory effects. This provokes the appearance of out of phase sinusoidal signals when the frequency of the input changes. Taking all the factors into account, the real output of a PA during a one tone characterization test follows the mathematical behavior represented by (4.27)

$$y_{NL}(t) = y_0 + \sum_{k>1}^{N} y_k cos(w_k t + \phi_k). \tag{4.17}$$

Identifying and measuring nonlinearities

In order to identify the real behavior of a device like an amplifier, there are several techniques that allow the engineer to get the real parameters that have been discussed within the section, such as gain, harmonics amplitudes or total harmonic distortion. Measurement techniques represent a key role in the design and verification or testing of PAs. Traditional ways to measure the non-linearity grade of a device under test (DUT) include power swept continuous waves to indentify the AM/AM or AM/PM distortion and the 1dB compression point or two tone measurements to find 3rd order intercept point [54]. Before talking about the different techniques to identify the behaviour of a PA, it is interesting to remark to most important measurements that help us on our task of measuring accurately the nonlinearities:

- **Amplitude to amplitude modulation characteristic (AM/AM):** This graph represents the amplitude of the fundamental tone of the output signal depending on the amplitude of input signal [55]. It can be observed that the PA starts with a linear behavior but reaches to a point where the output amplitude starts clipping. This effect is illustrated in Fig. 4.1.
- **Amplitude to phase modulation (AM/PM):** In contrast to the previous one, this graph illustrates the relation between the input amplitude modulation and the output phase modulation [56]. It is worth mentioning that, although the AM/AM characteristic can be observed in every nonlinear device, phase distortion effects are only observed in memory or dynamic devices such as PAs.

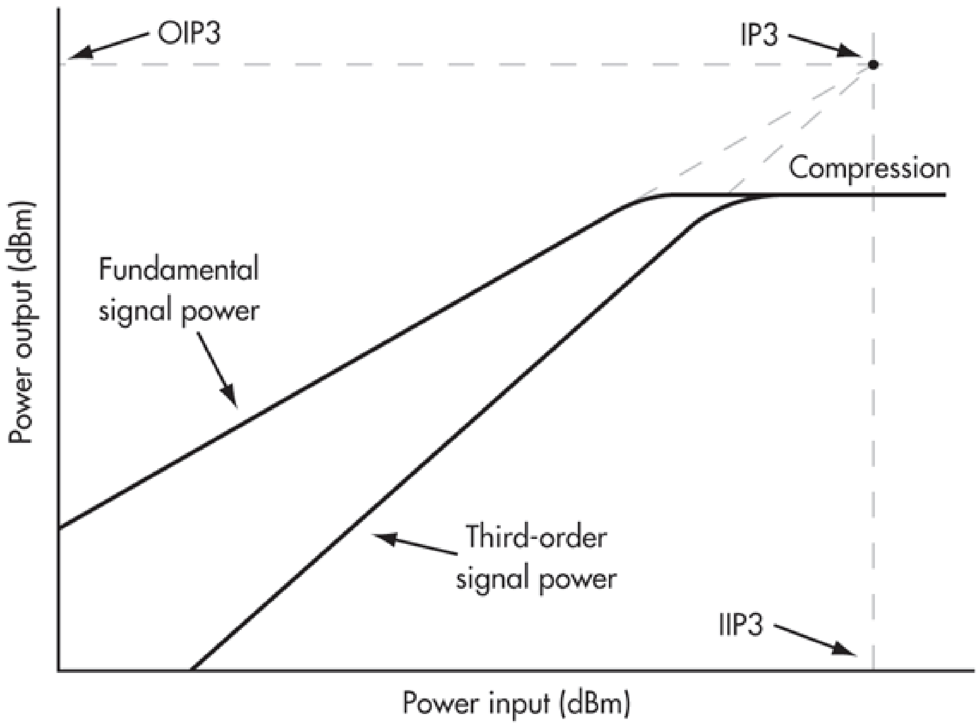

Figure 4.2 AM/AM characteristic [53].

- **1dB compression point:** It is a simple but effective way to measure the input power value where a PA starts behaving nonlinearly. It is defined as the point where the ideal output power is 1dB higher than the actual output power due to the clipping effect that has already been discussed during this section.
- **Total Harmonic Distortion (THD):** This measurement computes the percentage of output power that is produced by harmonics due to distortion in relation with the total output power. It is calculated as expressed in Eq.(4.6).
- **Error Vector Magnitude (EVM):** It is a figure of merit of the modulation quality. It is normally used with multi-symbol modulation methods [57]. A graphical representation of EVM is provided at Fig. 4.3.
- **Adjacent-Channel Power Ratio (ACPR):** This metric is particularly helpful to measure spectral regrowth, since it is computed as the ratio between the total power of signal measured at a lower or upper channel and the power in the main channel [59].

One-tone characterization

When an input signal mathematically expressed in Eq.(4.22) is set to a pure sinusoidal is used to identify the properties of a system in relation with distortion or spectral efficiency among others, the technique is named one tone characterization. This technique is the most simple one when trying to understand a system behavior. Although some effects cannot be observed by using one tone characterization, others are easier to identify in comparison with other techniques. The input signal is just like the one described in Eq.(4.22), but without amplitude or phase variation through time: $A(t) = A_i, \theta(t) = 0$. Therefore, considering the nonlinear effects that are responsible for the emergence of harmonics, amplitude compression and phase distortion, the output signal of a real PA when a one tone characterization technique is used can be mathematically described as

$$y_{NL}(t) = \sum_{k>0}^{\infty} A_{ok}(f,A_i) cos[w_k t + \phi_{ok}(f,A_i)], \tag{4.18}$$

where the phase and amplitude dependencies on the input frequency and amplitude has been taking into account and thus, they provoke the system to behave nonlinearly, since a linear system output would reproduce an output as following

$$y_L(t) = A_o cos[wt + \phi_o]. \tag{4.19}$$

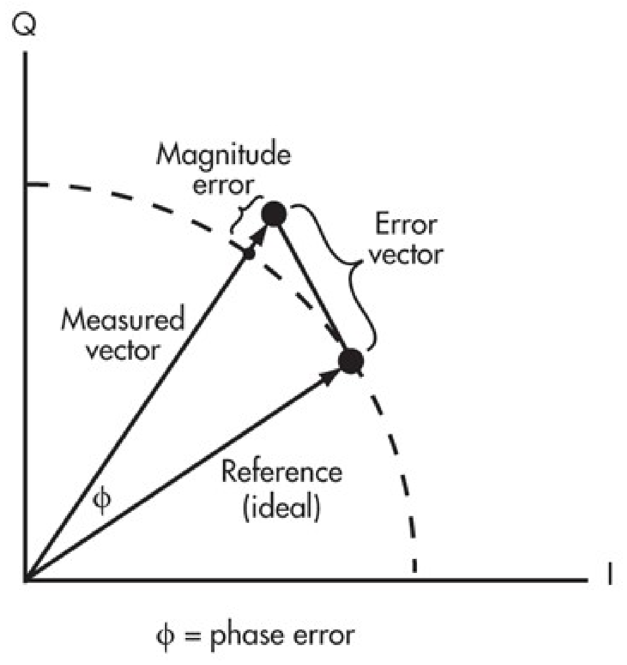

Figure 4.3 EVM representation [58].

Two-tone characterization

Two-tone characterization is a widely known technique to identify the AM/AM and AM/PM distortion. It consists in feeding the DUT with a signal based the summation of two different tones such as

$$x(t) = A_1 cos(w_1 t) + A_2 cos(w_2 t). \tag{4.20}$$

The nonlinear behavior of the PA will produce new frequency components at the output. Therefore, the output of the DUT can be expressed as

$$y_{NL}(t) = \sum_{i=1}^{\infty} A_{oi} cos(w_i t + \phi_{0i}), \tag{4.21}$$

where $w_i = mw_1 + nw_2$ being n,m two integers. These new frequency components will appear at every possible linear combination of the frequencies which were introduced as input to the DUT. They can be grouped in two different categories:

- **Harmonic distortion:** Their origin and effects have already been discussed. They do not appear due to interaction between the two tones. They appear at new frequency components at integer multiples of each of the fundamental tones.
- **Intermodulation:** This problem would not emerge in a one tone characterization test, since it happens due to the interaction between the two tones provided as input [60]. This means that intermodulation must only be taken into account when the input signal consists on multiple carriers. Intermodulation makes real PAs fail on accomplishing the principle of superposition, which was previously introduced. The appearance of new tones as a result of a linear combination of the input tones is an undesired effect on communication systems due to in band and out of band distortion:
  - **In-band distortion:** This concept has already been discussed when odd order harmonics were analyzed. They appear when $m+n=1$ and, considering the fact that both m and n are integers, it is easy to conclude that it continues happening when dealing with odd order components, if we compute the order as $|m|+|n|$. When this requirement is satisfied, the new components appear in the transmission band, which makes it harder to remove them. This new components

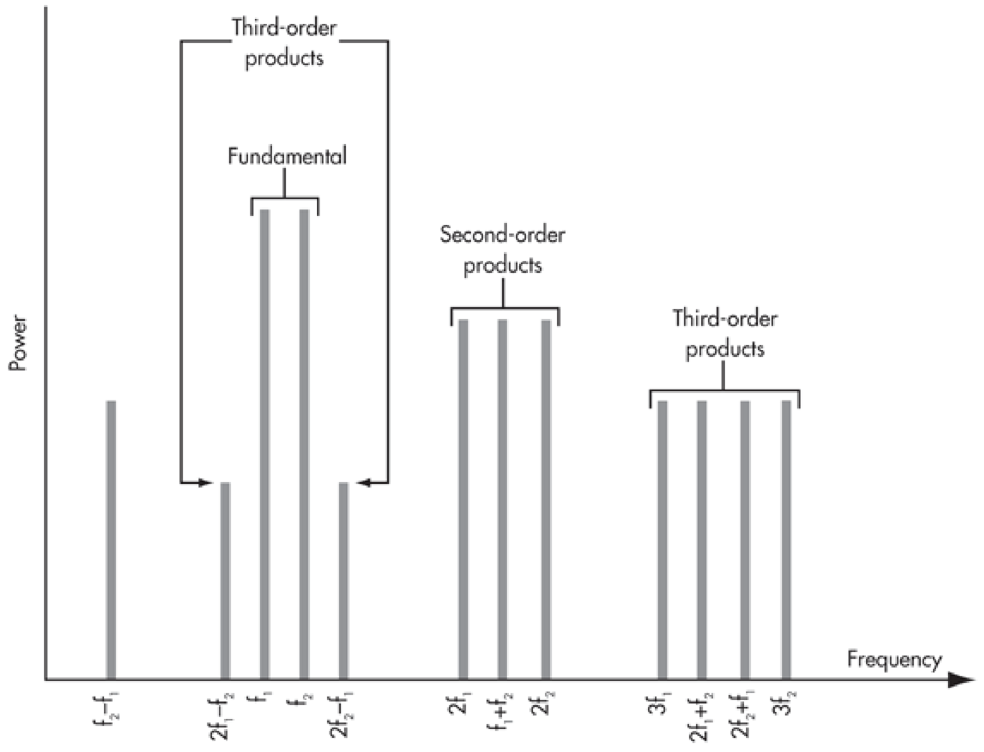

Figure 4.4 Intermodulation effects [51].

are named *n* order intermodulation products (IMn) and they are illustrated in Fig. 4.4. At this point, it is important to outline the *n* order intercept point measurements as other indicators of the system nonlinear characteristic. These points are defined as the place where the *n* order intermodulation power extrapolation overpasses the fundamental tone power extrapolation. The slope of the output-input power extrapolation graph is constant and equal to 1dB/dB. The third order intermodulation output power extrapolation, on the other hand, increases with 3dB/dB slope. Each *n* order intermodulation power extrapolation slope is equal to *n*. That is why, at some point (called IP3) both lines will reach the same place in the graph. A graphical representation of this effect is provided in Fig.4.2. Therefore, what determines the grade of linearity of a PA is not the slope of their IPn lines, since they are constant, but the point where they start, since this will establish where they will overpass the output power line. A basic magnitude that computes the grade of in-band distortion of a system is called error vector magnitude. It computes the quality of the system by calculating how accurately the PA is transmitting a constellation. It is computed as the root mean square value of the individual error vectors of every symbol in the constellation. The intermodulation distortion prdoducts can be clearly observed using a spectrum analyzer, although it only provides magnitude information. Phase information can be obtained by using a vector network analyzer.

– **Out-of-band distortion:** This type of distortion is way less problematic since, as it happens when $m+n \neq 1$, the intermodulation components appear outside the transmission band or in the DC component so, as it happens with even order harmonics, they can be removed with filtering [61].

Despite having presented intermodulation and harmonic distortion as problems for communication systems, it is worth mentioning that, in other fields such as the music, they can be deliberately applied in order to change the power spectral density of the output signal by filtering some output tones and making their amplitude stronger or lower than others and thus change the sound.

Memory Effects

Ideal systems are thought to have always the same response to the same input. However, as it usually happens, the past affects the present much more than we think. That is why, despite being ideally considered

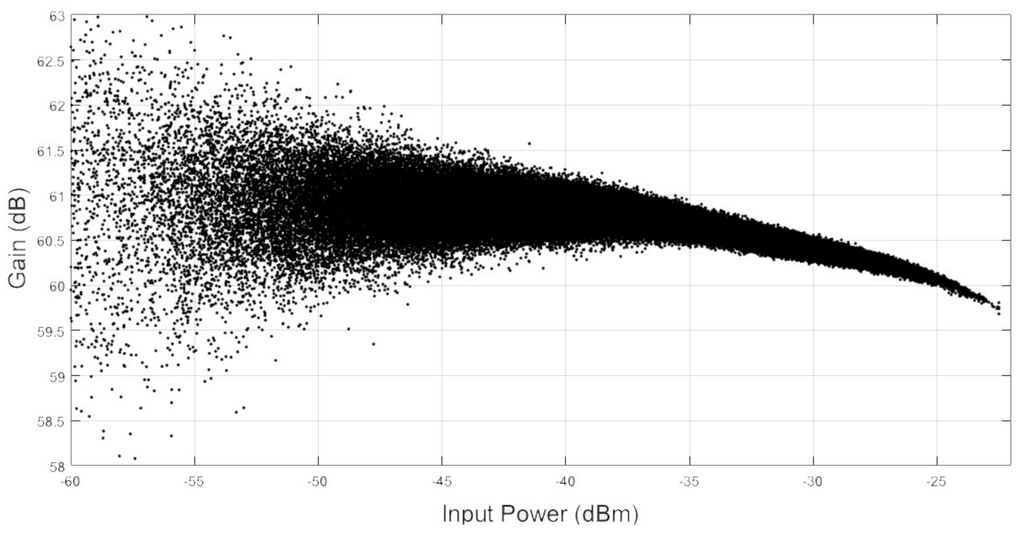

Figure 4.5 AM/Gain characteristic.

memoryless systems, the fact is that the previous inputs of a PA affect the current output. Therefore, having a current discrete input such as $x[n]$, we would be naive to believe that the real output could be expressed as a function that depends just on this input such as $y = f(x[n])$. A more realistic approach to express the output of a PA taking the memory effect into account would be $y = f(x[n], x[n-1], x[n-2] \ldots x[n-Q])$. Having considered this fact, it is worth mentioning that, as it happens in nature, the more close the past is to the present, the more it affects it. Therefore, it is easy to deduct that, although the output of a PA should theoretically depend on every previous input, the truth is that this effect gets less relevant the more we dwell on the past. This decrease is known as fading memory and the number of previous input samples that are taken into account as considerable influence the the current output represent the *memory depth*. During the previous section, phase distortion effects were already introduced. As it was stated, phase distortion may only occur in memory systems, since this type of distortion is related to how the system responses in memory terms when frequency changes. Basic electronic devices such as capacitors and coils are known to be able to store energy and thus change the current that flows in a circuit depending on how much energy they have storaged. Therefore, it should not be a surprise to understand that, if a simple R-C circuit is able to store energy and therefore has memory abilities, PAs also present this effects. Memory effects are responsible for the dynamic behavior of a system. This means that the system response changes through time. However, these variations can be grouped in two different timescales. These two timescales are short-term and long-term memory effects. While the first one occur with a similar rate as the input signal, the others appear with a slower frequency, closer to the modulation rate:

- **Short-term memory (STM) effects:** They happen with a frequency which similar to the carrier of the input signal. They are mostly produced by the matching networks, which can be reactive components of the transistor itself or external matching circuits of the printed-circuit board, and device capacitances. A good estimation of this effects can be obtained by measurement or simulation of the small signal frequency response of a system.
- **Long-term memory (LTM) effects:** As they name indicates, their timescale has nothing to do with the period of the carrier. This timescale is similar to the one of the envelope or modulation signal. They can be grouped in three different effects depending on the cause:
  - **Thermal effects:** Thermal energy storage and evolution occur in a slower timescale than the carrier or the envelope of the input signal. They appear with modulation frequencies lower than 100 kHz.
  - **Charge trapping:** They appear due to the imperfections of semiconductor materials. They are not likely to happen due to the high precision of the device fabrication process but when they do they can provoke variations of kHz or even MHz order.
  - **Bias circuit effects:** The bias line is used in order to provide an unimpeded flow of current DC while avoiding any RF signal to go thorough it, as it the circuit behaved as an open circuit for RF.

However, this ideal usage of the bias line results in high or low impedance for RF or DC. The low impedance that is intentionally provided to the DC component is maintained over a frequency range that receives the name of video bandwidth. Some intermodulation products such as IM2 or IM4 may appear at the video bandwidth and therefore, they will be less reduced than the others.

The easiest way to illustrate memory effects is in the AM/AM and AM/PM curves. As it was previously mentioned, memory effects are responsible for the dynamic behavior of a PA, so the same input results in different outputs each time it is introduced to the device, depending on the previous inputs as well. These so-called dispersion effects are clearly visible in Fig. 4.5, where gain compression is also evident. This figure was extracted from experimental measurements in a lab environment.

4.2.2 Power efficiency

When PAs, also known as large signal amplifiers, are used to handle large-voltage signals they achieve high energy efficiency levels. This is a critical aspect since energy saving is a major challenge for most of the modern technologies and considering the fact that PAs are one of the most energy consuming elements (nearly 40 per cent of the total energy consumption) in a wireless communications system, so efficiency is essential for them. Despite of this efficiency, PAs are inherently non-linear in this high power zone. Taking both of these facts into account, we find the conflict of interests that was introduced at the beginning of the chapter. On the one hand, efficiency is really important for a considerable energy consuming element such as the PA. On the other hand, linearity is needed in order to obtain a accurate output to a specific input and thus accomplish low error rates in the communication system. Before we get deeper into this conflict and how to deal with it, it is necessary to understand correctly a few concepts that we have already been talking about. The efficiency of a PA is defined as the ratio between the power of the output and the input as shown in Eq.(4.22)

$$\eta(\%) = \frac{P_1}{P_{DC}}. \tag{4.22}$$

Regarding the requirements that were described in the previous section, an ideal power amplifier should present a 100% power efficiency. A more accurate way to measure the efficiency of a PA is the power-added efficiency (PAE) defined as

$$PAE = \frac{P_1 - P_{in}}{P_{DC}} = \frac{P_1}{P_0}\left(1 - \frac{1}{G}\right), \tag{4.23}$$

but when the gain is high enough, both of the measurements are similar.

4.3 Behavioral modeling and digital predistortion

Mobile telecommunication techniques are speacially sentitive to spectral interferences and nowadays, lots of different communication technologies share the radio spectrum. That is the reason why there are regulatory organisations all over the world, whose job consists in ensuring that the spectral bandwidth division is not violated. In addition to this, modern applications need higher and higher bit rates every day. Both of these facts create the need of accumulating as much information as possible in a narrow spectral bandwidth, which is achieved by highly efficient modulation schemes. These techniques use multi-level combinations of amplitude and phase modulation [47] and thus have a high peak to average power ratio (PAPR) which is defined as the division between the maximum power of a transmitted symbol of modulation technique and the average power of transmission of a symbol as expressed in Eq.(4.10)

$$PAPR = \frac{P_{max}}{P_{avg}}. \tag{4.24}$$

If we wanted our amplifier to be completely linear, even when the peak power symbol is being transmitted, we would have to operate with it at a low input power. When a amplifier is working in this linear low power zone, we say that the PA is backed-off. This would solve the non-linearity issue but efficiency would drop dramatically since power efficiency is smaller with small input signals. This last fact makes it interesting for engineers to operate with PAs at stronger input power zones, where efficiency is better since less power percentage is dissipated. If we used a PA in a non-linear zone, we would need to solve somehow the previously discussed drawbacks of nonlinearities. This is where digital predistortion techniques appear as a effective solution [62]. Other techniques such as feedforward have been studied in order to avoid the spectral regrowth produced by nonlinearities [63]. Feedforward techniques are used directly at the output to remove distortion but, as it happens with the alternative of working with a backed-off PA, these workaround solve the problem but they are not efficient enough. Digital predistortion (DPD) is one of the most effective linearization techniques that make it possible to work with a PA in the compression zone while getting a linear behavior [64]. This technique transform the input of the power amplifier so that the output is closer to the desired output of an ideal amplifier. DPD can be understood as an inverse transformation of the non-desired effects of the PA before they appear at the output [65]. Therefore, when the signal arrives to the PA, this effects are canceled. DPD requires a model of the PA behavior in the zone where it is operating in order to cancel the unintended non-linear effects [66]. Fig. 4.6 illustrates a block diagram where the process of DPD is represented for ease of understanding.

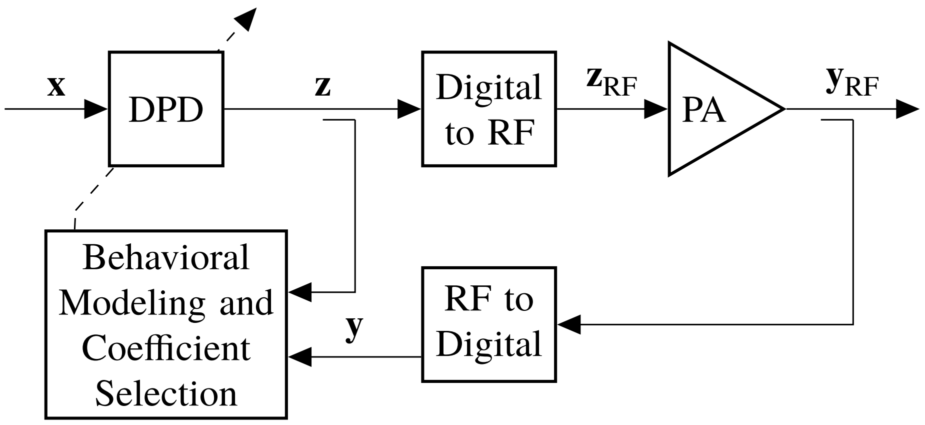

Figure 4.6 DPD block diagram.

4.3.1 Nonlinear System Modeling

Speaking on terms of voltage, a nonlinear memoryless polynomial system will behave as following:

$$V_{out}(t) = \sum_{i=1}^{\infty} A_i V_{in}^i(t) \tag{4.25}$$

The target of the digital preprocessing technique is to obtain an output as

$$V_{out}(t) = G(F_{PD}(V_{in}(t))) = A_1 V_{in}(t). \tag{4.26}$$

This is achieved by adding terms to the input in order to make the output look like the desired output. In order to know which terms are needed to be added to the input, DPD techniques need an accurate model of the system behavior. If we were dealing with static systems, an static predistorter would be enough to do the input-output mapping and solve efficiently the task. However, the dynamic behavior of PAs due to memory effects has already been discussed, so the predistortion needs to be as dynamic as the system itself. That is why predistorter coefficients values need to be adjusted as the system evolves.

Models and Power Amplifiers

The model of a system is a description of the relation between the inputs and outputs of that system. It is usually represented by a mathematical abstraction of that input-output relationship. It is important to understand that the efficiency of a model will depend not only on its accuracy to reproduce the real system behavior but also on its computational cost. A common approach to the modeling task is a physically based model. It is based on the usage of the physic properties of the devices that form the system in order to obtain a transfer function that describes the system behavior. There are other models that are derived directly from the measured data. They are also known as table-based models. Behavioral models are simplified mathematical models that take into account only the features that really affect the system behavior [67]. Conversely, there is another approach that has become very popular during last years. This approach is not based on the physics of the system, but in general mathematical parametric equations. This approach is named behavioral modeling. Then the application is DPD, there are physical features of the PA that are not interesting nor necessary to

Figure 4.7 Vito Volterra portrait photography.

model its nonlinear behavior, such as the frequency response or DC component. DPD requires computational efficiency and accurate results and behavioral models are good at this efficiency-accuracy balance.

Among these different types of models, it is essential to highlight Volterra series as one of the most powerful mathematical tools to the creation of behavioral models that represent the nonlinear behavior of a PA [68]. Vito Volterra was an Italian mathematician who led the development of functional analysis. He introduced the concept of *functionals* in the 1880s, functions that depend on a continuous set of values of another function. Functionals led to the development of new math areas such as solutions for integral or derivative equations. However, Volterra's theory was not applied to nonlinear systems modeling until 1958, when Norbert Wiener, a professor at MIT, applied it to this issue and published his work. The conflict of interest between distortion and energy effiency has provoked the rediscovery of Volterra techniques [69]. Volterra series has recently represented a key role on RF power amplifiers behavioral modeling and DPD linearization [48]. During this chapter, different mathematical approaches will be provided in order to explain the Volterra model and the different behavioral models that have been derived from them.

Volterra Series Introduction

A system can been defined as a black-box that computes and output to a input signal, where both the output and the input signals are functions of time. This can be modeled as

$$y(t) = Tx(t). \tag{4.27}$$

Besides that, with its theory of functionals, Volterra proved that every functional can be expressed as an expansion such as

$$y(t) = \sum_{n\geq 1} y_n(t). \tag{4.28}$$

Afterwards, Wiener realized about the possible applications that this theory of functionals could have in the nonlinear wireless communication systems modeling. Therefore, he used Eq.(4.28) and applied the traditional communications system theories which introduced the concept of T being a linear operator that can be described as the convolution

$$y(t) = H_1x(t) = \int_{-\infty}^{\infty} h_1(\tau)x(t-\tau)d\tau \tag{4.29}$$

of the input $x(t)$ with h_1, which is a linear kernel or input response. Unifying both concepts of functional expansion and convolution, he concluded that n grade functionals of Eq.(4.28) could be expressed as

$$y_n(t) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} h_n(\tau_1, \tau_2, ..., \tau_n)x(t-\tau_1)x(t-\tau_2)...x(t-\tau_n)d\tau_1 d\tau_2...d\tau_n. \tag{4.30}$$

It is worth mentioning that there is evidence of an important decrease in the relation between the n grade functional and $y(t)$ when n becomes increases. That why the summation in Eq.(4.28) is usually shortened as

$$y(t) = \sum_{n=1}^{N} y_n(t). \tag{4.31}$$

Volterra series is the result of unifying Eq(4.30) and Eq.(4.31). As it can be observed the are able to take into account every memory effect that leads to the different combinations of the actual and previous inputs and multiply each of those combinations with a kernel $h_n(\tau_1, \tau_2, ..., \tau_n)$ that represents the importance of that combination in the output of the system. Finally, it is important to remark the fact that we have done this development while considering analog signals. However, nowadays it is usual to work with digital signals, this mean that the time domain is not considered continuous, but discrete. That is why it is necessary to redefine the expressions developed in this chapter with the purpose of making them compatible with time discrete signals as

$$y[k] = \sum_{n=1}^{N} y_n[k] = \sum_{n=1}^{N} \sum_{q_1=0}^{Q} \sum_{q_n=0}^{Q} h_n(q_1, q_2, ..., q_n) \prod_{j=1}^{n} x(k - q_j). \tag{4.32}$$

Now that the Volterra series has been introduced, several mathematical approaches which can be very helpful to analyze and understand them will be presented.

Volterra Series as a development of Taylor Series This development is intended to show how a Volterra Series can be deducted from a nonlinear instantaneus model [70]. Let's start from a nonlinear polynomial expression such as

$$y(t) = a_0 + \sum_{n=1}^{N} a_n x^n(t), \tag{4.33}$$

where the polynomial coefficients a_n are computed thanks to the Taylor series of the input-output relation expanded at x_0. This point is named operating point andn it is defined by the supply or DC voltage. This expansion results in (4.34)

$$\begin{aligned} y(x(t)) &= y(x)\big|_{x=x_0} + \frac{1}{2!}\frac{\partial y}{\partial x}\bigg|_{x=x_0}(x - x_0) + \frac{1}{3!}\frac{\partial^2 y}{\partial x^2}\bigg|_{x=x_0}(x - x_0)^2 + ... \\ &= a_0 + a_1 x(t) + a_2 x^2(t) + ... \end{aligned} \tag{4.34}$$

However, it is trivial to understand that this equation does not consider dynamic effects, since the output only depends in the current input. Therefore, we need to introduce memory term dependency so that $y \neq f(x)$ but $y = f(x,x_1,x_2,x_3...,x_n)$ where $x_i = x(t - \tau_i)$. This would let us rewrite Eq.(4.34) with memory effects being taken into account as

$$\begin{aligned} y(x(t)) = y(x)\big|_{x=x_0} &+ \frac{1}{2!}\frac{\partial y}{\partial x}\bigg|_{x=x_0}(x - x_0) + \frac{1}{2!}\frac{\partial y}{\partial x_1}\bigg|_{x=x_0}(x_1 - x_0) + ... \\ &+ \frac{1}{3!}\frac{\partial^2 y}{\partial x^2}\bigg|_{x=x_0}(x - x_0)^2 + \frac{1}{3!}\frac{\partial^2 y}{\partial {x_1}^2}\bigg|_{x=x_0}(x_1 - x_0)^2 + ... \\ &+ \frac{1}{3!}\frac{\partial^2 y}{\partial x \partial x_1}\bigg|_{x=x_0}(x - x_0)(x_1 - x_0) + ... \end{aligned} \tag{4.35}$$

which is a multinomial series. If enough time is taken to observe this equation, it can be noticed that the memory terms such as $(x_1 - x_0)$ or $(x - x_0)(x_1 - x_0)$ can be also found in Eq.(4.32), being named the latter as a Volterra cross-term. Therefore, as it has been deducted, a Volterra series can be understood as a Taylor series generalization, with memory effects dependencies being introduced.

Volterra Series as a development of a linear system Another interesting approach that needs to be presented in this section is starting from linear system. Linear systems do not need to be memoryless. As a matter of fact, the output of a dynamic linear system can be expressed as the convolution of the system transfer response with the input signal as

$$y(t) = h(t) * x(t) = \int_0^t h(\tau)x(t-\tau)d\tau. \tag{4.36}$$

The idea of using higher order of the input signal in order to model a nonlinear behavior has already been used several times within this section and that is why it should not be a surprise that this concept was applied to Eq.(4.36). If we did this with a second order nonlinearity while still modeling memory effects, Eq.(4.36) could be rewritten as

$$y_2 = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} h_2(\tau_1,\tau_2)x(t-\tau_1)x(t-\tau_2)d\tau_1 d\tau_2, \tag{4.37}$$

where each signal has its own delay parameter. Now, if we extend the nonlinear model development to its n-th order, we end up having an expression like

$$y_n(t) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} h_n(\tau_1,\tau_2,...,\tau_n)x(t-\tau_1)x(t-\tau_2)...x(t-\tau_n)d\tau_1 d\tau_2...d\tau_n. \tag{4.38}$$

Although the integral limits have been changed to $(-\infty,\infty)$, it has only been done due to generalization purposes, since the systems that are being considered are causal and time invariant. Therefore, in order to build the complete model we would need a summation of the N terms as expressed in Eq.(4.31). Once again, Volterra kernels $h_n(\tau_1,\tau_2,...,\tau_n)$ can be observed as the coefficients of the model that allow to balance the importance of each of the terms in the development. This development gives evidence to state that a linear convolution is just a particular case of Volterra series for $n=1$.

Power Amplifiers Behavioral Modeling based on Volterra Series

Volterra series have been lately the most popular tool at behavioral models constructing. However, since it was introduced as an infinite series, it is impossible to work with. Therefore, models will need to cut this series at some point. There have been presented different workarounds to cut this series while still considering the most important components that have led to the different models that are derived from Volterra series [71]. New technological developments have led to new types of PA models such as ANN-based ones [72], a comparison between their performance at DPD can be found in [73]. Indeed, ANNs have been proven to perform accurate results at Volterra kernels extraction [31]. Before starting with model definition and understanding the differences between them, it is worth mentioning that, at modeling problems, it is typical to work with the complex envelope of the input instead of the input itself. Complex envelope of a signal is defined as

$$\tilde{x} = A(t)e^{j\phi(t)s}, \tag{4.39}$$

although it will be mentioned as $x(t)$ in order to simplify notation.

Full Volterra Model (FV) The complete Volterra model or FV [74] is the most direct implementation from the Volterra Series to a model when considering a complex envelope as an input, the model is mathematically expressed as

$$\begin{aligned} y_{FV}[k] = \sum_{p=1}^{P}\sum_{q_1=0}^{Q}\sum_{q_2=q1}^{Q}\cdots\sum_{q_p=q_p-1}^{Q}\sum_{q_p+1=q_p}^{Q}\sum_{q_p+2=q_p+1}^{Q}\cdots \\ \cdots\sum_{q_{2p-1}=q_{2p-2}}^{Q} h_{2p-1}(q_1,\ldots,q_{2p-1})X\prod_{i=1}^{p}x[k-q_i]\prod_{j=p+1}^{2p-1}x^*[k-q_j]. \end{aligned} \tag{4.40}$$

The first thing that can be observed about this model is the fact that the regressors order is always odd, being $2P-1$ the maximum order. Since we are working with the complex envelope of the signal, there have to be $p-1$ conjugated factors and p non-conjugated ones in order to make $y[k]$ and $x[k]$ have the same fundamental tone. The reason why each summation depends on the value of the parameter of the previous one is to avoid repetitions of combinations that would lead to considering a regressor more than once. Finally, it is important to consider that thanks to the fading memory effect, the Volterra series can be transformed to a sufficiently accurate finite model. Namely, due to the decrease of the regressor importance when n increases, Q represents the *memory depth* of the model.

Generalized Memory Polynomial Also known as GMP [75], this model comes from the memory polynomial model (MP) [65], so it is worth understanding MP model before getting to know the GMP. Memory polynomial can be seen as a simple and elegant approach to introduce memory effects in a static nonlinear polynomial model. Its behavior is reproduced by Eq.(4.41)

$$y_{MP} = \sum_{n=1}^{N} \sum_{m=0}^{M} a_{nm} x[k-q_m] |x[k-q_m]|^{n-1} \tag{4.41}$$

It is just a simplification of Volterra series in which one there are left only diagonal regressors. There are not Volterra cross memory terms. Therefore, we cannot find terms such as $x(n)x(n-1)$. This provokes the fact that each polynomial only depends on a certain delay of the input signal, not a combination of several different delays. The assumption is that non-diagonal terms do not contribute enough to make it computationally efficient to compute their kernels, since they are not relevant on accuracy enhancement. However, some upper and lower diagonal terms may contribute enough to make it worth it for a model to consider them. That is why the Generalized Memory Polynomial was presented as an approach to avoid all those non diagonals terms from being removed to the model.

$$\begin{aligned} y_{GMP}[k] = &\sum_{n=1}^{N} \sum_{m=0}^{M} a_{nm} x[k-q_m] |x[k-q_m]|^{n-1} \\ &\sum_{n=1}^{N} \sum_{m=0}^{M} \sum_{sm=0}^{S} b_{nms} x[k-q_m] |x[k-q_m-q_s]|^{n-1} \\ &\sum_{n=1}^{N} \sum_{m=0}^{M} \sum_{sm=0}^{S} c_{nms} x[k-q_m] |x[k-q_m+q_s]|^{n-1}. \end{aligned} \tag{4.42}$$

As the name indicates, this model contains the memory polynomial model. In fact, MP model is reproduced by the first summation. Besides the MP, the GMP introduces another parameter S, which allows the upper and lower diagonal regressors to appear and have its own b_{nms} or c_{nms} kernels. That is how this shorter, and thus more computationally efficient model than FV is defined.

5 Random Forest Application to Volterra Kernels Selection

I don't want to belong to any club that will accept me as a member

Groucho Marx

This chapter presents the application of the Random Forest technique to behavioral modeling component selection. Several digital predistorters (DPDs) with a random structure are attained to obtain a sequence of the regressors importance with respect to their impact in the linearization error of a power amplifier (PA). The approach has been validated in the DPD of a commercial PA operating under a 5G-NR signal, showcasing the ability of the algorithm to sort and prune the components.

5.1 Introduction

The evolution of digital communication standards, such as those included in the fifth generation (5G) of mobile communications, has led to new challenges about power amplifiers (PAs) performance. Orthogonal frequency division multiplexing (OFDM) has become one of the most used modern modulation techniques due to its narrowband spectral efficiency. Despite having interesting advantages, this modulation method is characterized by its high peak-to-average power ratio (PAPR), which is understood as its significant drawback. Since PAs are the most power-consuming element of a wireless communication system, improving their efficiency is essential to avoid energy wastage. Considering the importance of energy savings and its environmental effects, it is a widespread practice to work with PAs at their peak power, i.e., in their saturation zone, where their maximum power efficiency is achieved. These nonlinearities lead to undesired consequences such as in-band distortion and spectral regrowth.

Achieving power efficient wireless communication systems while maintaining its linear behavior has become a challenging issue in the industry [76]. Digital predistortion (DPD) techniques arise as a solution to this efficiency-linearity trade-off. DPD has become one of the most popular PA linearization techniques due to its excellent performance [77]. However, in order to compensate the unwanted nonlinearities of the PA, an accurate model of its behavior is required.

Among the different behavioral models that have been presented with this aim, Volterra-series-based models have been highlighted as promising candidates thanks to their accurate performance on modeling nonlinear systems with memory. Considering that Volterra series are affected by the *curse of dimensionality*, that is, their number of components grow rapidly with the nonlinear order and memory depth, component selection techniques appear to reduce the computational requirements without losing significant accuracy. Different pruning techniques have emerged in this framework [**?**]. These regression methods are based on selecting the most relevant coefficients of a model under a specific situation, e.g., a fixed power level or signal characteristics like bandwidth and modulation, therefore they present an inherent lack of generalization ability. Within the supervised learning techniques, random forests [78] are known to be good at generalization since these are ensemble learning methods based on aggregating results of different situations. In the context of model order reduction techniques for DPD, Random Forest can be classified as a brute force selection

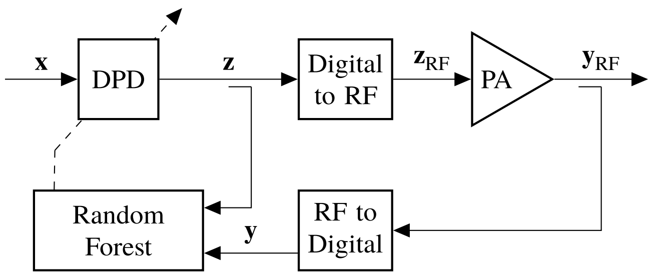

Figure 5.1 Random Forest application to DPD.

Table 5.1 Input data to the Random Forest algorithm where 1 and 0 stands for the inclusion of the i-th regressor ϕ_i into the model..

t	ϕ_1	ϕ_2	$\dots$	ϕ_n	NMSE (dB)
1	0	1	$\dots$	1	-45.8
2	0	0	$\dots$	1	-41.1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
N	1	1	$\dots$	0	-35.9

technique, together with less relevant basis removal [79]. The application of a Random Forest algorithm to DPD is illustrated in Fig. 5.1.

In this work, we focus on the application of random forest to determine the structure of a PA model. This contribution continues with the theoretical framework in Section 5.2. Theory of random forest is provided in Section 5.3 along with the experimental part in Section 5.4. Section 5.5 concludes this work.

5.2 Volterra Series and Linear Regression

Linear algebra methods such as multiple linear regression are often used to compute Volterra-series model parameters. Thus, the input-output signal relationship is modeled through the measurement equation $\mathbf{y} = \mathbf{X}\mathbf{h} + \varepsilon$ where $\mathbf{y} \in \mathbb{C}^m$ represents a m-sample segment of the system response, the observation matrix $\mathbf{X} \in \mathbb{C}^{m \times n}$ contains the n model regressors, and $\mathbf{h}$ represents the model coefficients, also known as Volterra kernels, and ε is the measurement noise. An estimation of $\mathbf{h}$ is computed by using the least squares solution as $\hat{\mathbf{h}} = (\mathbf{X}^H \mathbf{X})^{-1}\mathbf{X}^H\mathbf{y}$. Once the Volterra kernels have been obtained, an estimate of the system response is calculated as $\hat{\mathbf{y}} = \mathbf{X}\hat{\mathbf{h}}$. In a DPD scenario, the model is employed to obtain a predistorted signal $\mathbf{z}$. Eventually, in order to evaluate the DPD performance, the linearization normalized mean squared error (NMSE) [80] is computed as $\text{NMSE} = \frac{\|\mathbf{x}\mathbf{G}-\mathbf{y}\|_2^2}{\|\mathbf{x}\mathbf{G}\|_2^2}$, when $\mathbf{z}$ is applied as the input, G represents the gain of the PA and $\|\cdot\|_2$ stands for the ℓ_2 norm of its argument.

Volterra kernels estimation provides accurate results for specific situations such as an invariant PA operating point, but it does not achieve good results at generalization. Due to its well known generalization capability, a Random Forest approach, where Volterra series models are considered as individual decision trees of which results are averaged, is presented as a promising workaround.

5.3 Random Forest

Decision trees are useful and simple machine learning models which are characterized by a hierarchical structure based on recursively dividing the input on subsets. At the root node, it begins considering the whole dataset, and selects the best split variable among all of them. This process, which is called *induction* [81], is repeated until a split is found for each of the variables. The goal of induction is to create subsets formed of similar data. There are different approaches to achieve this target. For ease of understanding, we will start considering binary decision trees, where variables can only value 0 or 1. In order to obtain homogeneous subsets, a simple technique is to split on a variable and measure the variance of the output when the value of this variable is fixed [82]. The variable with the lowest output variance will be considered as the next one to use as split. Hence, variance will be computed as:

$$var(j) = w_0 var(s_0) + w_1 var(s_1), \tag{5.1}$$

where w_i is the amount of samples that belong to the s_i subset of the output, which accomplish that the j-th variable is equal to i. When working with non-binary decision trees, the only difference of this method is that a previous step needs to be made. At binary regression trees, there is no need to worry about where to split the values of each variable, since there are only two different values. However, when variables are not binary, the first step requires finding the best split point at each variable. This is usually made by going over the whole range of values of the variable, dividing the output dataset and comparing the variance computed as described in Eq.(5.1) until the lowest is found. This induction process continues until the maximum number of splits (known as *depth* of the tree) is reached. Other parameters such as the minimum number of samples in each subset are also used to limit the iterations of the induction algorithm. Otherwise, induction would continue until every variable had been used as split. After induction, a pruning technique may be applied to the tree. Pruning is used to remove splits that do not improve the accuracy of the tree, reducing in this way the complexity of the model. In addition to this, pruning is useful to reduce overfitting, which is a common problem of this machine learning algorithm. Various pruning techniques have been studied [83] and they are classified in post-pruning and pre-pruning. The latter is not done after induction, but at the same time. One of the most simple but yet effective post-pruning methods is Reduced Error Pruning. It considers every split node as possible point of pruning. It consists of removing the subtree rooted at that node, making it a leaf node and evaluating the performance of the pruned tree. If its accuracy is better or equal to the original accuracy, the pruned tree will replace the original one.

As it was previously discussed, decision trees tend to overfit, so they are highly sensitive to input variations. Ensemble learning algorithms, which have recently become popular due to their flexibility and steadiness, have solved this drawback by generating models and aggregating their results [84]. Two methods were early derived from this concept: *boosting* and *bagging* of decision trees. Bagging methods are based on the construction of trees that take different random subsets from the input, named as bootstrap datasets. These trees are completely independent between each other so that they can be trained in parallel. At bagging, all the data has the same probability to appear in a new bootstrap sample. However, at boosting, new trees focus on the data that was wrongly classified or regressed by previous trees [85]. Hence, trees need to be sequentially created, since every tree depends on the success of the previous ones.

Random Forests are based on bagging of decision trees, so from now on we will focus on this technique. After trees are independently constructed, usually without pruning, they are used for classification, where the most voted class is considered as the result, or regression, where the output model is used to obtain an estimate. Volterra series models can be considered as decision trees where a split represents the consideration of a regressor into the model, as it is illustrated in Fig. 5.2.

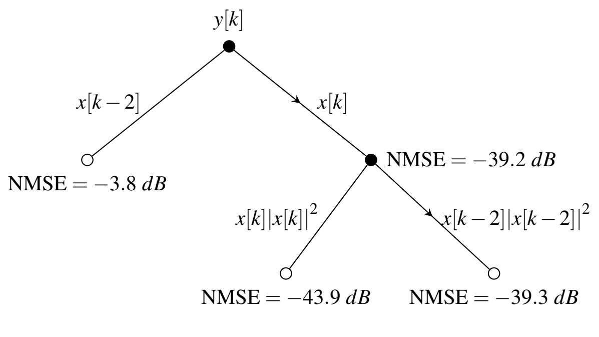

Figure 5.2 Decision tree representation of a Volterra series model, in which $x[k]$ is chosen at the root leaf and $x[k]|x[k]|^2$ at the second split. At each leaf node the attained normalized mean square error (NMSE) is given as an example..

Decision trees which form a Random Forest take a bootstrap dataset and, at each split, not every predictor is considered on the best split selection, but a random subset of them. This new random step provides additional robustness to the model. The performance of this model is usually evaluated by computing the out-of-bag (OOB) error as described in Algorithm 1 [86]. This indicator represents the average NMSE for each input x_i sample, which is computed using predictions from the trees did not use x_i during their induction or training stage. After that, these trees are used for classification, where the most voted class is considered as the result,

Algorithm 1 Random Forest

```
Input: $\mathbf{X} \in \mathbb{C}^{m \times n}$, $y \in \mathbb{C}^m$, $N_T$ = number of trees
Output: $Imp$
 1: Initialization :
 2: for $model = 1$ to $N_T$ do
 3:     Estimate a model of $n/2$ random regressors.
 4:     Calculate linearization NMSE.
 5: end for
 6: for $tree = 1$ to $N_T$ do
 7:     Over 2/3 of the the length of the input signal:
 8:     for $split = 1$ to $n$ do
 9:         Randomly select $n/3$ regressors of the model.
10:         Split in the regressor with the lowest variance.
11:     end for
12:     Obtain the validation NMSE of the tree.
13: end for
14: Compute the OOB error as $\sum_{t=1}^{N_T} \frac{NMSE_t}{N_T}$
15: for $f = 1$ in $n$ do
16:     Shuffle the values of feature $f$ while leaving the rest unchanged to obtain $X_f^{\psi}$.
17:     Get predictions of $y$ using $X_f^{\psi}$ as input.
18:     Compute the importance of this variable as:
19:  $\mathrm{Imp}(f) = \mathrm{NMSE}(X_f^{\psi}) - \mathrm{OOB}$
20: end for
```

or regression, where the output model is used to obtain an estimate. Besides these applications, Random Forests achieve to compute accurate variable importance by randomly permuting the values of a predictor while leaving the others unchanged, making new predictions and computing the new out-of-bag error as it is expressed in Eq. (5.2) [87]. The difference between this new OOB and the one that was computed by using the unchanged original input sample as represents the variable importance, also known as permutation *importance*. In terms of the problem at hand, variable importance estimation can be used to get the relevance of each one of the n regressors on predicting y as

$$\mathrm{Imp}(f) = \mathrm{OOB}(X_f^{\psi}) - \mathrm{OOB}(X), \tag{5.2}$$

where X_f^{ψ} represents the input matrix with randomly permuted values at the f-th feature column. Volterra series models can be considered as decision trees where a split represents the consideration of a regressor into the model, as it is illustrated in Fig. 5.2. Repeating this process for every single variable would result in a feature importance vector Imp, from which pruning decisions can be made.

Table 5.2 DOMP and Random Forest DPD linearization performance in a sweep of number of coefficients.

#	**DOMP**				**Random Forest**			
coeff.	NMSE (dB)	ACPR-1(dBc)	ACPR+1(dBc)	EVM(%)	NMSE (dB)	ACPR-1(dBc)	ACPR+1 (dBc)	EVM(%)
5	-40.0	-49.7	-50.8	1.2	-41.8	-49.5	-50.3	1.0
6	-40.4	-50.7	-51.9	1.2	-42.5	-50.5	-51.4	1.0
7	-40.9	-54.2	-54.6	1.1	-43.0	-52.2	-53.3	1.0
8	-41.1	-54.6	-55.2	1.1	-43.6	-52.5	-54.0	0.9
9	-46.8	-54.4	-54.9	0.8	-45.7	-52.5	-54.0	0.8
10	-46.9	-54.5	-54.9	0.8	-46.5	-52.5	-54.2	0.8
15	-46.8	-54.2	-55.2	0.8	-47.3	-54.0	-53.7	0.8
20	-48.4	-54.4	-54.8	0.8	-48.5	-54.8	-54.5	0.8
25	-48.5	-54.8	-54.3	0.8	-48.4	-54.7	-54.0	0.8

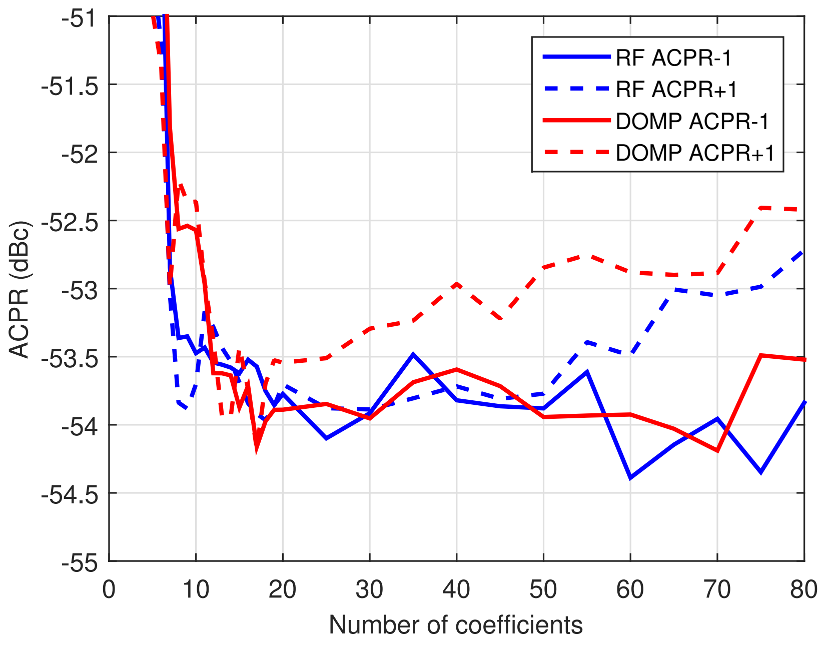

Figure 5.3 ACPR attained by Random Forest and DOMP..

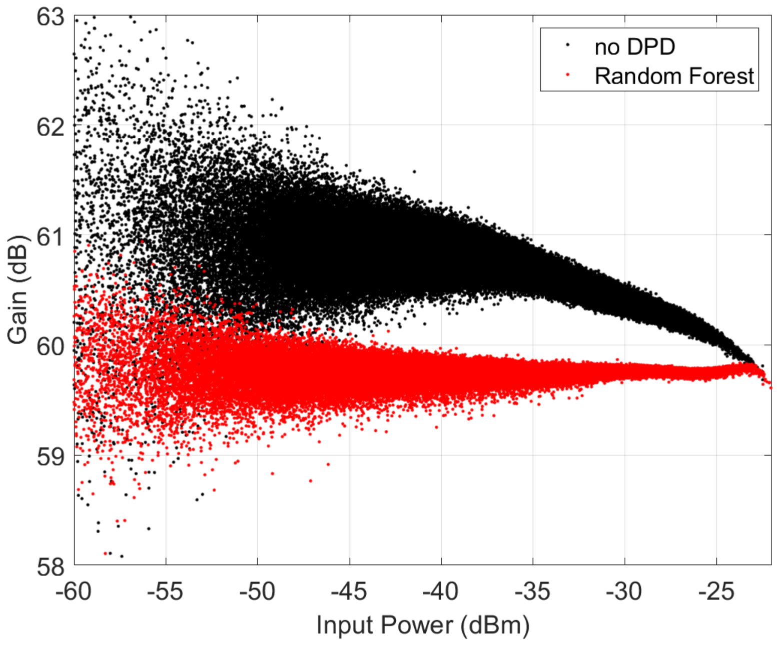

Figure 5.4 AM/Gain characteristics of a PA without DPD and with DPD composed of the 25 most relevant coefficients identified by Random Forest..

5.4 Experimental Validation

The validation of the presented technique was executed over a 30 MHz 5G-NR signal acquired on the testbench described in [88]. The PA operating point was fixed to +27.5 dBm of average output power which corresponds to 1.2 dB of gain compression. The input dataset was built by using 5000 DPD models obtained by randomly pruning a generalized memory polynomial (GMP) of 13th order and maximum memory depth of 15. Each one of them was applied to one OFDM symbol and the linearization NMSE was used as the output vector. Table 5.1 shows the structure of the data that was introduced to a Random Forest. The Random Forest

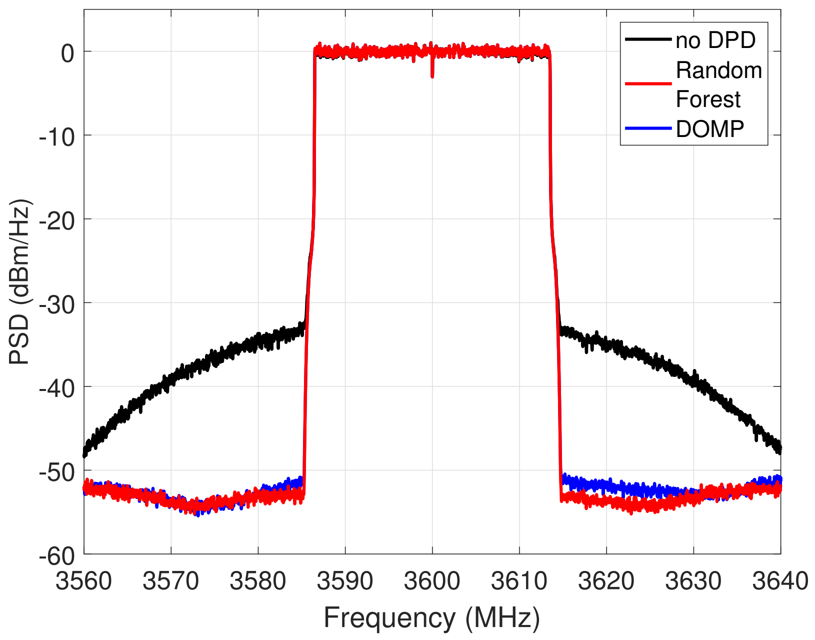

Figure 5.5 Power spectral density of the signal without DPD and with 25-coefficient DPDs attained with Random Forest and DOMP.

computed a vector in which the regressors are sorted in order of importance. Using this vector, a sequence of DPD models following the order of importance was applied. As a comparison, the list of regressors returned by the doubly orthogonal matching pursuit (DOMP) sorting technique [89] was also executed. Table 5.2 shows the performance attained for both algorithms varying the number of coefficients in a range from 5 to 25. Random Forest exhibits a better identification of the relevant coefficients in the low number of coefficients range. AM/Gain characteristics for the PA without DPD and with a DPD of 25 coeffients obtained with the Random Forest are illustrated in Fig. 5.4, where the linearization of the DUT is clearly observed. Fig. 5.5 shows the power spectral density in the same situation adding the results for the DOMP technique with the same number of coeffients, where a spectral regrowth reduction is observed with respect to the case without DPD, achieving a similar performance with both algorithms. Another figure of merit is presented in Fig 5.3 to show the spectral benefits achieved by Random Forest, where the upper and lower Adjacent Channel Power Ratio (ACPR) are illustrated.

5.5 Conclusion

In this work, the application of Random Forest to the determination of Volterra-based DPD models for PAs have been presented. Experimental results show how this groundbreaking technique is able to extract the most relevant coefficients within a DPD by processing a set of random-structured models. Random Forest promising generalization benefits are yet to be studied in future research.

res

6 Codes

Algorithm 6.1 Complex Valued Neural Network Implementation. This code is based on an implementation of a natural valued ANN, from which changes have been made to adapt it.

```
import numpy as np
import scipy.io as sio
import math
from scipy.spatial import distance
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings("error")

# Dataset loading
#dataset = sio.loadmat('y_lte15MHz_ovs6_35c5_CreeZHL42W_ppc')
dataset = sio.loadmat('syntheticsignal2')
#X = (np.random.rand(1000 , 1) + np.random.rand(1000 , 1)*1j) +1 +1j

X = dataset["x"]
Y = dataset["y"]
from sklearn.preprocessing import MinMaxScaler
sc = MinMaxScaler(feature_range = (1, 2))
X = sc.fit_transform(X)
Y = sc.transform(Y)
mem=0
train_size = 1000

X_train =X [0:train_size]
X_test = X



y_test= Y
for i in range( 1, mem+1):
    X_train = np.append(X_train, X[i:1000+i], 1)
    X_test = np.append(X_test, X[0+i:30000+i], 1)

y_train = Y [mem:train_size+mem]
# y_test = Y [0+mem:30000+mem]
p=mem+1
topology = [p ,4, 1]
learn = 200
```

```
class Layer ():
    # This class represents a Layer, formed by a group of neurons and weights
        between this layer and the previous one
    def __init__(self, conn, n_neur, act_f, der_f, der_f_vconj):
        self.neur=[]
        for i in range (0, n_neur):
            self.neur.append( Neuron (act_f(2*i+1), der_f(2*i+1), der_f_vconj(2*
                i+1)))
   #     self.b = (np.random.rand(1,neur) + np.random.rand(1,neur)*1j) * 2 -1
       -1j
        self.W = (np.random.rand(conn , n_neur) + np.random.rand(conn , n_neur)
            *1j) * 2 - 1 - 1j


class Neuron ():
    # Class that represents a neuron, formed by an activation function and its
        derivatives
    def __init__(self,act_f, der_f, der_f_vconj):
        self.act_f= act_f
        self.der_f=der_f
        self.der_f_vconj=der_f_vconj


# DEFINICION DE LAS FUNCIONES DE ACTIVACION Y SUS DERIVADAS RESPECTO A W

# CLMS en funcion del orden de la neurona


def clms_function (n):
    clms= lambda v: v * (np.absolute(v)**(n-1))
    return clms

def der_clms_v (n):
    der_clms_v = lambda v: ((n+1)/2) * (np.absolute(v)**(n-1))
    return der_clms_v


def der_clms_vconj (n):
    der_clms_vconj = lambda v : ((n-1)/2) * ((np.absolute(v)**(n-3))*(v**2))
    return der_clms_vconj

def nmse (y, yest):
    calc= 20*math.log10(np.linalg.norm(y-yest)/np.linalg.norm(y))
    return calc

# Model definition and creation

def create_ann (topology, act_f, der_f, der_f_vconj):
    ann = []

    for n_layer in range(1, len(topology)):
            ann.append(Layer(topology[n_layer-1],topology[n_layer], act_f, der_f
                , der_f_vconj ))
    return ann

#We use MSE as cost function
```

```
error = lambda y, yest : (y-yest)
f_cost = lambda y, yest :(1/2)*np.mean((np.absolute(error(y,yest)))**2)
der_fcost = lambda y, yest: np.absolute(y,yest)
neural_net = create_ann(topology, clms_function, der_clms_v, der_clms_vconj)

#Training

def train(neural_net, X, Y , der_cost_mse, lr=0.5, train = True):
    out = [(None,X)]
    #forward

    for n_layer in range(0, len(neural_net)):
        v =  out[-1][1] @ neural_net[n_layer].W #+ neural_net[n_layer].b
        y =  np.zeros([ len (X[:]),len(neural_net[n_layer].neur)]) + np.zeros([
            len (X[:]),len(neural_net[n_layer].neur)])*1j
        for n_neur,neur in enumerate(neural_net[n_layer].neur):
            y[:,n_neur] = neur.act_f(v[:,n_neur])
        out.append((v, y))

    #backward
    if train:
      #  lr = np.full((n))
        deltas = []
        errores = []
        for n_layer in reversed(range(0, len(neural_net))):
            v = out[n_layer+1][0]
            y = out[n_layer+1][1]

            der_f_vconj= np.zeros([ len (X[:]),len(neural_net[n_layer].neur)]) +
                 np.zeros([ len (X[:]),len(neural_net[n_layer].neur)])*1j
            der_f_v = np.zeros([ len (X[:]),len(neural_net[n_layer].neur)]) + np
                .zeros([ len (X[:]),len(neural_net[n_layer].neur)])*1j
            lrs = []
            for n_neur , neur in enumerate(neural_net[n_layer].neur):
                der_f_vconj[:,n_neur] = neur.der_f_vconj(v[:, n_neur])
                der_f_v[:, n_neur] = neur.der_f((v[:, n_neur]))
                lrs.append(lr*(100**(-2*np.linalg.norm(der_f_v[:,n_neur])))) #
                    esto provoca resultados interesantes, probar con señal
                    sintetica.
               #lrs.append(lr*(10000**n_neur))

            if n_layer == len (neural_net) -1:
                delta= +0.5*( (np.conj(error(Y,y))* der_f_vconj)+ (error(Y,y)*
                    der_f_v)) #tested
                deltas.insert (0, delta)
                errores.insert(0, np.conj(error(Y,y))*der_f_v )
                errores.insert(1, error(Y,y)*der_f_vconj)
                errores.insert(2, np.conj(error(Y,y))*der_f_vconj)
                errores.insert(3, error(Y,y)*der_f_v)




            else:
                a=errores[0] @ _W.T
                b= errores[1] @ _W.T
                c = errores[2] @ np.conj(_W.T) #si hubiera mas capas habria que
                    establecer mejor la relacion
                d = errores[3] @ np.conj(_W.T)
                delta = 0.5 *((a+b)* der_f_v +(c+d)*der_f_vconj)
                deltas.insert (0,delta)
            _W = neural_net[n_layer].W
                # Gradient descent
                # Update equation
            neural_net[n_layer].W = neural_net[n_layer].W + np.conj(out[n_layer
                ][1]).T @ deltas[0] * lr # Incremento de W es lr*delta*salida de
                 esa capa
           # print( neural_net[n_layer].W)
            #print (der_f_vconj[0])
    return out[-1][1]



# Visualization (test and training)

import time
from IPython.display import clear_output

neural_net = create_ann(topology, clms_function, der_clms_v, der_clms_vconj)

loss = []
nmses= []
for i in range(0, 600000):

  # Let's train
  pY = train(neural_net, X_train, y_train, der_fcost , lr=learn)

  if i % 100 == 0:
   # print(pY)
    print("Error conseguido durante el entrenamiento = ")
    last_loss=f_cost(y_train, pY)
    loss.append(last_loss)
    print (last_loss)
    clear_output(wait=True)
    y_pred = train(neural_net
                 , X_test, y_test , der_fcost, lr=0.0001, train = False)
    NMSE = nmse(y_test, y_pred)
    nmses.append(NMSE)
    plt.plot(range(len(nmses)), nmses)
    plt.show()
    time.sleep(0.5)
    print("Valor del NMSE conseguido = ")
    print (NMSE)
```

Algorithm 6.2 Script that does the data preprocessing, calls the other scripts and presents the results.

```
%% INPUT PREPROCESSING
clear all;
tic;
load('random_forest_v2');
x=xsw;
y=ymed;
Nseg = 1;
n_trees=1500;
```

```
regresores = string([1:1:81]);
regresores= "R" + regresores;
models=[];
n_regs=40;
n_iters = [5000];
Mdl =[];
val=false;
i=1;
parfor iseg = 1:5000
    dBm = @(x) 10*log10(rms(x).^2/100)+30;
    dBminst = @(x) 10*log10(abs(x).^2/100)+30;
    PAPR = @(x) 20*log10(max(abs(x))/rms(x));
    maxdBm = @(x) dBm(x) + PAPR(x);
    scale_dBm = @(x,P) x*10^((P-dBm(x))/20);
    model_PA.tipo = 'GMP';
    model_PA.extension_periodica = 0;
    model_PA.grafica = 0;
    model_PA.h = [];
    model_PA.pe = 0;
    model_PA.type = 'GMP';
    model_PA.extension_periodica = 0;
    model_PA.grafica = 0;
    model_PA.h = [];

    model_PA.Ka = [0:12];
    model_PA.La = 15*ones(size(model_PA.Ka));
    model_PA.Kb = [2 4 6];
    model_PA.Lb = [1 1 1];
    model_PA.Mb = [1 1 1];
    model_PA.Kc = [2 4 6];
    model_PA.Lc = [1 1 1];
    model_PA.Mc = [1 1 1];
    model_PA.dc = 0;
    model_PA.cs = 0;
    model_PA.nmax = 200;

    warning ('off','stats:robustfit:RankDeficient');
    warning ('off','MATLAB:nearlySingularMatrix');
    orden = [1:81];
    orden = orden(randperm(length(orden)));
    orden = orden(1:n_regs);
    model_PA.s = orden;
    model_PA.calculo = 'shuffle';
    ind = [1:2*length(x)/10];
    %ind = [(iseg-1)*length(x)/Nseg+1:iseg*length(x)/Nseg];
    %disp(fprintf('%d: desde %d a %d',iseg,(iseg-1)*length(x)/Nseg+1,iseg*
        length(x)/Nseg));
    disp(fprintf('Iteracion %d',iseg));
    x_id = x(ind);
    y_id = y(ind);
    x_va = x(end/2:end);
    y_va = y(end/2:end);

    model_PA = model_gmp_domp_omp(y_id, x_id, model_PA,val)

    models=[models;model_PA];
    model_PA=[];
    %si la h no esta vacia, el modelo se valida, no identifica uno nuevo.
    %[model_PA] = model_gmp_domp_omp(y_va, x_va, model_PA)
end

    n_comb=n_iters;
    A=zeros(n_comb,81); %si no se usa pondremos un 0, si se usa, un uno
    NMSE=[];
    for i = 1:n_comb
        A(i, models(i).s)=1; %el resto cero porque no se usan
        NMSE(i,1)= models(i).nmse;
    end
T = array2table(A,'VariableNames',cellstr(regresores));
NMSE = array2table(abs(NMSE));
T= [T, NMSE];
%%
X = NMSE;
Y= A;
[n,m]=size(X);
[Xtrain, idtrain] = datasample(X, round(0.67*n));
Ytrain= Y(idtrain);
idtest      = 1:n;
idtest(idtrain) = [];
X_test        = X(idtest);
Y_test = Y(idtest);

imp = manual_importances(T(1:81),T)
[B,I] = maxk(imp, length(imp));
regresores_order= "Regresor " + models(1).Rmat(I);
O= table(regresores_order, B', 'VariableNames',{'RegresoresOrder', 'Importance'
    });
orden = I;
configuration;
model_PA.s = orden(1:n_regs);
model_PA.calculo = 'shuffle';
val = true;
model_PA = model_gmp_domp_omp(y_id, x_id, model_PA, true);
toc
```

Algorithm 6.3 Script that implements the Random Forest importance estimation algorithm.

```
%funcion de calculo de las importancias de las variables a partir de la
%matriz T (indica los regresores incluidos en cada modelo aleatorio y el
%NMSE conseguido)
function [importances] = manual_importances(X, y)
[Xtrain, idtrain] = datasample(X, round(0.66*n));
idtest      = 1:n;
idtest(idtrain) = [];
X_test        = X(idtest);
for train_idx, test_idx in range(10):
    X_train, X_test = X[train_idx], X[test_idx]
    Y_train, Y_test = Y[train_idx], Y[test_idx]
    r = manual_randomforest.fit(X_train, Y_train)
    acc = r2_score(Y_test, rf.predict(X_test))
    for i in range(X.shape[1]):
        X_t = X_test.copy()
        np.random.shuffle(X_t[:, i])
        shuff_acc = r2_score(Y_test, rf.predict(X_t))
        scores[names[i]].append((acc-shuff_acc)/acc)
print ("Features sorted by their score:")
print (sorted([(round(np.mean(score), 4), feat) for
             feat, score in scores.items()], reverse=True))
end
    end
end
end
```

Algorithm 6.4 Random Forest class. It is used for importance estimation.

```
class RandomForest():
    def __init__(self, x, y, n_trees, n_features, sample_sz, depth=10, min_leaf
        =5):
        np.random.seed(12)
        if n_features == 'sqrt':
            self.n_features = int(np.sqrt(x.shape[1]))
        elif n_features == 'log2':
            self.n_features = int(np.log2(x.shape[1]))
        else:
            self.n_features = n_features
        print(self.n_features, "sha: ",x.shape[1])
        self.x, self.y, self.sample_sz, self.depth, self.min_leaf = x, y,
            sample_sz, depth, min_leaf
        self.trees = [self.create_tree() for i in range(n_trees)]

    def create_tree(self):
        idxs = np.random.permutation(len(self.y))[:self.sample_sz]
        f_idxs = np.random.permutation(self.x.shape[1])[:self.n_features]
        return DecisionTree(self.x.iloc[idxs], self.y[idxs], self.n_features,
            f_idxs,
                    idxs=np.array(range(self.sample_sz)),depth = self.depth,
                        min_leaf=self.min_leaf)

    def predict(self, x):
        return np.mean([t.predict(x) for t in self.trees], axis=0)

def std_agg(cnt, s1, s2): return math.sqrt((s2/cnt) - (s1/cnt)**2)

class DecisionTree():
    def __init__(self, x, y, n_features, f_idxs,idxs,depth=10, min_leaf=5):
        self.x, self.y, self.idxs, self.min_leaf, self.f_idxs = x, y, idxs,
            min_leaf, f_idxs
        self.depth = depth
        print(f_idxs)
#         print(self.depth)
        self.n_features = n_features
        self.n, self.c = len(idxs), x.shape[1]
        self.val = np.mean(y[idxs])
        self.score = float('inf')
        self.find_varsplit()

    def find_varsplit(self):
        for i in self.f_idxs: self.find_better_split(i)
        if self.is_leaf: return
        x = self.split_col
        lhs = np.nonzero(x<=self.split)[0]
        rhs = np.nonzero(x>self.split)[0]
        lf_idxs = np.random.permutation(self.x.shape[1])[:self.n_features]
        rf_idxs = np.random.permutation(self.x.shape[1])[:self.n_features]
        self.lhs = DecisionTree(self.x, self.y, self.n_features, lf_idxs, self.
            idxs[lhs], depth=self.depth-1, min_leaf=self.min_leaf)
        self.rhs = DecisionTree(self.x, self.y, self.n_features, rf_idxs, self.
            idxs[rhs], depth=self.depth-1, min_leaf=self.min_leaf)

    def find_better_split(self, var_idx):
        x, y = self.x.values[self.idxs,var_idx], self.y[self.idxs]
        sort_idx = np.argsort(x)
        sort_y,sort_x = y[sort_idx], x[sort_idx]
        rhs_cnt,rhs_sum,rhs_sum2 = self.n, sort_y.sum(), (sort_y**2).sum()
        lhs_cnt,lhs_sum,lhs_sum2 = 0,0.,0.

        for i in range(0,self.n-self.min_leaf-1):
            xi,yi = sort_x[i],sort_y[i]
            lhs_cnt += 1; rhs_cnt -= 1
            lhs_sum += yi; rhs_sum -= yi
            lhs_sum2 += yi**2; rhs_sum2 -= yi**2
            if i<self.min_leaf or xi==sort_x[i+1]:
                continue

            lhs_std = std_agg(lhs_cnt, lhs_sum, lhs_sum2)
            rhs_std = std_agg(rhs_cnt, rhs_sum, rhs_sum2)
            curr_score = lhs_std*lhs_cnt + rhs_std*rhs_cnt
            if curr_score<self.score:
                self.var_idx,self.score,self.split = var_idx,curr_score,xi

    @property
    def split_name(self): return self.x.columns[self.var_idx]

    @property
    def split_col(self): return self.x.values[self.idxs,self.var_idx]

    @property
    def is_leaf(self): return self.score == float('inf') or self.depth <= 0


    def predict(self, x):
        return np.array([self.predict_row(xi) for xi in x])

    def predict_row(self, xi):
        if self.is_leaf: return self.val
        t = self.lhs if xi[self.var_idx]<=self.split else self.rhs
        return t.predict_row(xi)
```

Algorithm 6.5 Decision tree class. A Random Forest is based on independently generated decision trees.

```
%% INPUT PREPROCESSING
clear all;
tic;
load('random_forest_v2');
```

```
x=xsw;
y=ymed;
Nseg = 1;
n_trees=1500;
regresores = string([1:1:81]);
regresores= "R" + regresores;
models=[];
n_regs=40;
n_iters = [5000];
Mdl =[];
val=false;
i=1;
parfor iseg = 1:5000
    dBm = @(x) 10*log10(rms(x).^2/100)+30;
    dBminst = @(x) 10*log10(abs(x).^2/100)+30;
    PAPR = @(x) 20*log10(max(abs(x))/rms(x));
    maxdBm = @(x) dBm(x) + PAPR(x);
    scale_dBm = @(x,P) x*10^((P-dBm(x))/20);
    model_PA.tipo = 'GMP';
    model_PA.extension_periodica = 0;
    model_PA.grafica = 0;
    model_PA.h = [];
    model_PA.pe = 0;
    model_PA.type = 'GMP';
    model_PA.extension_periodica = 0;
    model_PA.grafica = 0;
    model_PA.h = [];

    model_PA.Ka = [0:12];
    model_PA.La = 15*ones(size(model_PA.Ka));
    model_PA.Kb = [2 4 6];
    model_PA.Lb = [1 1 1];
    model_PA.Mb = [1 1 1];
    model_PA.Kc = [2 4 6];
    model_PA.Lc = [1 1 1];
    model_PA.Mc = [1 1 1];
    model_PA.dc = 0;
    model_PA.cs = 0;
    model_PA.nmax = 200;

    warning ('off','stats:robustfit:RankDeficient');
    warning ('off','MATLAB:nearlySingularMatrix');
    orden = [1:81];
    orden = orden(randperm(length(orden)));
    orden = orden(1:n_regs);
    model_PA.s = orden;
    model_PA.calculo = 'shuffle';
    ind = [1:2*length(x)/10];
    %ind = [(iseg-1)*length(x)/Nseg+1:iseg*length(x)/Nseg];
    %disp(fprintf('%d: desde %d a %d',iseg,(iseg-1)*length(x)/Nseg+1,iseg*
        length(x)/Nseg));
    disp(fprintf('Iteracion %d',iseg));
    x_id = x(ind);
    y_id = y(ind);
    x_va = x(end/2:end);
    y_va = y(end/2:end);
    model_PA = model_gmp_domp_omp(y_id, x_id, model_PA,val)

    models=[models;model_PA];
    model_PA=[];
    %si la h no esta vacia, el modelo se valida, no identifica uno nuevo.
    %[model_PA] = model_gmp_domp_omp(y_va, x_va, model_PA)
end

    n_comb=n_iters;
    A=zeros(n_comb,81); %si no se usa pondremos un 0, si se usa, un uno
    NMSE=[];
    for i = 1:n_comb
        A(i, models(i).s)=1; %el resto cero porque no se usan
        NMSE(i,1)= models(i).nmse;
    end
T = array2table(A,'VariableNames',cellstr(regresores));
NMSE = array2table(abs(NMSE));
T= [T, NMSE];
%%
X = NMSE;
Y= A;
[n,m]=size(X);
[Xtrain, idtrain] = datasample(X, round(0.67*n));
Ytrain= Y(idtrain);
idtest      = 1:n;
idtest(idtrain) = [];
X_test       = X(idtest);
Y_test = Y(idtest);

imp = manual_importances(T(1:81),T)
[B,I] = maxk(imp, length(imp));
regresores_order= "Regresor " + models(1).Rmat(I);
O= table(regresores_order, B', 'VariableNames',{'RegresoresOrder', 'Importance'
    });
orden = I;
configuration;
model_PA.s = orden(1:n_regs);
model_PA.calculo = 'shuffle';
val = true;
model_PA = model_gmp_domp_omp(y_id, x_id, model_PA, true);
toc
```

List of Figures

List of Tables

List of Codes

Bibliography

[1] Perry Sadorsky. Information communication technology and electricity consumption in emerging economies. *Energy Policy*, 48:130–136, sep 2012.

[2] M. M. A. Hossain and Riku Jantti. Impact of efficient power amplifiers in wireless access. In *2011 IEEE Online Conference on Green Communications*. IEEE, sep 2011.

[3] Martha C. Paredes Paredes and M. Julia Fernández-Getino García. The problem of peak-to-average power ratio in ofdm systems.

[4] Allen Katz, John Wood, and Daniel Chokola. The evolution of pa linearization: From classic feedforward and feedback through analog and digital predistortion. *Microwave Magazine, IEEE*, 17:32–40, 02 2016.

[5] Ultrawideband Digital Predistortion (DPD): The Rewards (Power and Performance) and Challenges of Implementation in Cable Distribution Systems. *https://www.analog.com/en/analog-dialogue/articles/ultrawideband-digital-predistortion-rewards-and-challenge-of-implementation-in-cable-system.html*.

[6] Alberto Prieto, Beatriz Prieto, Eva Martinez Ortigosa, Eduardo Ros, Francisco Pelayo, Julio Ortega, and Ignacio Rojas. Neural networks: An overview of early research, current frameworks and new challenges. *Neurocomputing*, 214:242–268, Nov 2016.

[7] Olivier Bousquet, Ulrike von Luxburg, and Gunnar Rätsch, editors. *Advanced Lectures on Machine Learning*. Springer Berlin Heidelberg, 2004.

[8] Oludare Isaac Abiodun, Aman Jantan, Abiodun Esther Omolara, Kemi Victoria Dada, Nachaat AbdElatif Mohamed, and Humaira Arshad. State-of-the-art in artificial neural network applications: A survey. *Heliyon*, 4(11):e00938, Nov 2018.

[9] Haohan Wang and Bhiksha Raj. On the origin of deep learning.

[10] Alan L. Wilkes and Nicholas J. Wade. Bain on neural networks. *Brain and Cognition*, 33(3):295–305, Apr 1997.

[11] Donald O. Hebb. *The organization of behavior: A neuropsychological theory*. Wiley, New York, June 1949.

[12] Computational Performance Evolution. *https://www.top500.org/statistics/perfdevel/*.

[13] Erkki Oja. Simplified neuron model as a principal component analyzer. *Journal of Mathematical Biology*, 15(3):267–273, Nov 1982.

[14] Teshome Alemu. *Recognition of Amharic Braille*. PhD thesis, Mar 2009.

[15] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, Jan 2015.

[16] Alberto Prieto, Miguel Atencia, and Francisco Sandoval. Advances in artificial neural networks and machine learning. *Neurocomputing*, 121:1–4, Dec 2013.

[17] Ivars Namatēvs. Deep convolutional neural networks: Structure, feature extraction and training. *Information Technology and Management Science*, 20, Dec 2017.

[18] Daniel Motta, Alex Álisson Bandeira Santos, Ingrid Winkler, Bruna Aparecida Souza Machado, Daniel André Dias Imperial Pereira, Alexandre Morais Cavalcanti, Eduardo Oyama Lins Fonseca, Frank Kirchner, and Roberto Badaró. Application of convolutional neural networks for classification of adult mosquitoes in the field. *PLOS ONE*, 14(1):e0210829, jan 2019.

[19] Understanding CNNs. *https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/*.

[20] Recurrent Neural Networks Overview. *https://towardsdatascience.com/recurrent-ne/*.

[21] Understanding LSTMs. *https://towardsdatascience.com/understanding-lstm-and-its-quick-implementation-in-keras-for-sentiment-analysis-af410fd85b47*.

[22] Yuxi Li. Deep reinforcement learning: An overview.

[23] Introduction to Reinforcement Learning. *https://towardsdatascience.com/introduction-to-various-reinforcement-learning-algorithms-i-q-learning-sarsa-dqn-ddpg-72a5e0cb6287*.

[24] Joe Bible, Susmita Datta, and Somnath Datta. Cluster analysis. In *Informatics for Materials Science and Engineering*, pages 53–70. Elsevier, 2013.

[25] What is Reinforcement Learning? *https://deepsense.ai/what-is-reinforcement-learning-the-complete-guide/*.

[26] AI index annual reports. *http://aiindex.org/*.

[27] J.C. Pedro and S.A. Maas. A comparative overview of microwave and wireless power-amplifier behavioral modeling approaches. *IEEE Transactions on Microwave Theory and Techniques*, 53(4):1150–1163, apr 2005.

[28] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 2(4):303–314, dec 1989.

[29] V.Z. Marmarelis and X. Zhao. Volterra models and three-layer perceptrons. *IEEE Transactions on Neural Networks*, 8(6):1421–1433, 1997.

[30] Simon S. Haykin. *Neural networks and learning machines*. Pearson Education, Upper Saddle River, NJ, third edition, 2009.

[31] Jelena Misic, Vera Markovic, and Zlatica Marinkovic. Volterra kernels extraction from neural networks for amplifier behavioral modeling. oct 2014.

[32] GridSearchCV Reference Website. *https://scikit-learn.org/stable/modules/grid_search.html*.

[33] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer New York, 2009.

[34] K-Fold Cross-Validation. *https://towardsdatascience.com/why-and-how-to-cross-validate-a-model-d6424b45261f*.

[35] Underfitting versus Overfitting at Regression. *https://docs.aws.amazon.com/machine-learning/latest/dg/model-fit-underfitting-vs-overfitting.html*.

[36] Overfitting and Underfitting depending on Model Complexity. *https://machinelearningmastery.com/diagnose-overfitting-underfitting-lstm-models/*.

[37] Fenglei Fan and Ge Wang. Universal approximation with quadratic deep networks.

[38] Franco Manessi and Alessandro Rozza. Learning combinations of activation functions.

[39] Hans Georg Zimmermann, Alexey Minin, Victoria Kusherbaeva, and Muenchen Germany. Comparison of the Complex Valued and Real Valued Neural Networks Trained with Gradient Descent and Random Search Algorithms.

[40] N. Benvenuto, M. Marchesi, F. Piazza, and A. Uncini. A COMPARISON BETWEEN REAL AND COMPLEX VALUED NEURAL NETWORKS IN COMMUNICATION APPLICATIONS. In *Artificial Neural Networks*, pages 1177–1180. Elsevier, 1991.

[41] Louis L. Scharf Peter J. Schreier. *Statistical Signal Processing of Complex-Valued Data*. Cambridge University Press, 2014.

[42] Simone Scardapane, Steven Van Vaerenbergh, Amir Hussain, and Aurelio Uncini. Complex-valued neural networks with nonparametric activation functions. *IEEE Transactions on Emerging Topics in Computational Intelligence*, pages 1–11, 2018.

[43] A. Uncini, L. Vecci, P. Campolucci, and F. Piazza. Complex-valued neural networks with adaptive spline activation function for digital-radio-links nonlinear equalization. *IEEE Transactions on Signal Processing*, 47(2):505–514, 1999.

[44] Laurent Sorber, Marc Van Barel, and Lieven De Lathauwer. Unconstrained optimization of real functions in complex variables. *SIAM Journal on Optimization*, 22(3):879–898, jan 2012.

[45] N. Benvenuto and F. Piazza. On the complex backpropagation algorithm. *IEEE Transactions on Signal Processing*, 40(4):967–969, apr 1992.

[46] Nitzan Guberman. On complex valued convolutional neural networks.

[47] Andrei Grebennikov, Nathan O. Sokal, and Marc J. Franco. *Switchmode RF and Microwave Power Amplifiers*. 07 2012.

[48] Steven W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, San Diego, CA, USA, 1997.

[49] Robert L. Boylestad and Louis Nashelsky. *Electronic Devices and Circuit Theory (8th Edition)*. Prentice Hall, 2001.

[50] *Analog Circuit Design*. Elsevier, 1991.

[51] Third Order Intercept and 1dB compression point. *https://www.electronicdesign.com/what-s-difference-between/what-s-difference-between-third-order-intercept-and-1-db-compression-point*.

[52] Doron Shmilovitz. On the definition of total harmonic distortion and its effect on measurement interpretation. *Power Delivery, IEEE Transactions on*, pages 526 – 528, Feb. 2005.

[53] RF Power Amplifier Guide. *http://www.super-conn.com/content/42-A*.

[54] David Wisell. Measurement techniques for characterization of power amplifiers. 2007.

[55] P. M. Cabral, J. C. Pedro, and N. B. Carvalho. Dynamic am-am and am-pm behavior in microwave pa circuits. In *2005 Asia-Pacific Microwave Conference Proceedings*, volume 4, pages 4 pp.–, Dec 2005.

[56] G Acciari, Paolo Colantonio, M De Dominicis, and Massimiliano Rossi. A Fast AM/AM and AM/PM Characterization Technique. 05 2019.

[57] Fu-Ling Lin, Shin-Fu Chen, Liang-Fang Chen, and Huey-Ru Chuang. Computer simulation and measurement of error vector magnitude (evm) and adjacent-channel power ratio (acpr) for digital wireless communication rf power amplifiers. In *Gateway to 21st Century Communications Village. VTC 1999-Fall. IEEE VTS 50th Vehicular Technology Conference (Cat. No.99CH36324)*, volume 4, pages 2024–2028 vol.4, Sep. 1999.

[58] Understanding Error Vector Magnitude. *https://www.electronicdesign.com/engineering-essentials/understanding-error-vector-magnitude*.

[59] Peter Jantunen, Timo I. Laakso, and Teknillinen Korkeakoulu. Modelling of nonlinear power amplifiers for wireless communications. 2004.

[60] J. Carlos Pedro and Nuno Borges Carvalho. *Intermodulation distortion in microwave and wireless circuits*. Boston, MA : Artech House, 2003. Includes bibliographical references and index.

[61] Ali Cheatio. *Analytical analysis of in-band and out-of-band distortions for multicarrier signals : impact of nonlinear amplification, memory effect and predistortion*. PhD thesis, March 2017.

[62] Houssam Eddine Hamoud, Tibault Reveyrand, Sebastien Mons, and Edouard Ngoya. A comparative overview of digital predistortion behavioral modeling for multi-standards applications. In *2018 International Workshop on Integrated Nonlinear Microwave and Millimetre-wave Circuits (INMMIC)*. IEEE, Jul 2018.

[63] A. Zhu and T.J. Brazil. Behavioral modeling of RF power amplifiers based on pruned Volterra series. *IEEE Microwave and Wireless Components Letters*, 14(12):563–565, dec 2004.

[64] Mohamed K. Nezami. Pre-distortion fundamentals of power amplifier linearization using digital pre-distortion. 2004.

[65] J. Kim and K. Konstantinou. Digital predistortion of wideband signals based on power amplifier model with memory. *Electron. Lett.*, 37(23):1417, 2001.

[66] Ibrahim Can Sezgin. Different Digital Predistortion Techniques for Power Amplifier Linearization, 2016. Student Paper.

[67] Harald Enzinger. *Behavioral Modeling and Digital Predistortion of Radio Frequency Power Amplifiers*. PhD thesis, Marc 2018.

[68] Anding Zhu and Thomas J. Brazil. An Overview of Volterra Series Based Behavioral Modeling of RF/Microwave Power Amplifiers. In *2006 IEEE Annual Wireless and Microwave Technology Conference*. IEEE, dec 2006.

[69] C. Crespo-Cadenas, P. Aguilera-Bonet, J.A. Becerra-González, and S. Cruces. On nonlinear amplifier modeling and identification using baseband volterra-parafac models. *Signal Processing*, 96:401–405, mar 2014.

[70] S. Hassouna, P. Coirault, and T. Poinot. Non-linear system identification using volterra series expansion. *IFAC Proceedings Volumes*, 33(15):947 – 952, 2000. 12th IFAC Symposium on System Identification (SYSID 2000), Santa Barbara, CA, USA, 21-23 June 2000.

[71] Georgios Birpoutsoukis, Péter Zoltán Csurcsia, and Johan Schoukens. Efficient multidimensional regularization for volterra series estimation.

[72] M. Rubiolo, G. Stegmayer, and D. Milone. Compressing a neural network classifier using a Volterra-Neural Network model. jul 2010.

[73] Tomas Gotthans, Geneviève Baudoin, and Amadou Mbaye. Digital predistortion with advance/delay neural network and comparison with volterra derived models. 2015:811–815, Jun 2015.

[74] S. Benedetto, E. Biglieri, and R. Daffara. Modeling and Performance Evaluation of Nonlinear Satellite Links-A Volterra Series Approach. *IEEE Transactions on Aerospace and Electronic Systems*, AES-15(4):494–507, jul 1979.

[75] D.R. Morgan, Z. Ma, J. Kim, M.G. Zierdt, and J. Pastalan. A generalized memory polynomial model for digital predistortion of RF power amplifiers. *IEEE Transactions on Signal Processing*, 54(10):3852–3860, oct 2006.

[76] L. Guan and A. Zhu. Green communications: Digital predistortion for wideband RF power amplifiers. *IEEE Microw. Mag.*, 15(7):84–99, Nov 2014.

[77] C. Fager, T. Eriksson, F. Barradas, K. Hausmair, T. Cunha, and J. C. Pedro. Linearity and efficiency in 5G transmitters: New techniques for analyzing efficiency, linearity, and linearization in a 5G active antenna transmitter context. *IEEE Microw. Mag.*, 20(5):35–49, May 2019.

[78] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics New York, 2001.

[79] P. L. Gilabert, G. Montoro, T. Wang, M. N. Ruiz, and J. A. García. Comparison of model order reduction techniques for digital predistortion of power amplifiers. In *2016 46th European Microwave Conference (EuMC)*, pages 182–185, Oct 2016.

[80] D. Schreurs. *RF Power Amplifier Behavioral Modeling*. The Cambridge RF and Microwave Engineering Series. Cambridge University Press, 2008.

[81] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, Mar 1986.

[82] Hemant Ishwaran. Variable importance in binary regression trees and forests.

[83] Nikita S. Patel and Saurabh Upadhyay. Study of various decision tree pruning methods with their empirical comparison in weka. 2012.

[84] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, Oct 2001.

[85] Harris Drucker and Corinna Cortes. Boosting decision trees. volume 8, pages 479–485, 01 1995.

[86] Tom Bylander. Estimating generalization error on two-class datasets using out-of-bag estimates. *Machine Learning*, 48(1):287–297, Jul 2002.

[87] Gilles Louppe, Louis Wehenkel, Antonio Sutera, and Pierre Geurts. Understanding variable importances in forests of randomized trees. volume 26, 12 2013.

[88] C. Crespo-Cadenas, M. J. Madero-Ayora, and J. A. Becerra. Volterra-based behavioral modeling, parameter estimation, and linearization. In *2018 IEEE MTT-S Latin America Microwave Conference (LAMC 2018)*, pages 1–4, Dec 2018.

[89] J. A. Becerra, M. J. Madero-Ayora, J. Reina-Tosina, C. Crespo-Cadenas, J. García-Frías, and G. Arce. A doubly orthogonal matching pursuit algorithm for sparse predistortion of power amplifiers. *IEEE Microw. Wireless Compon. Lett.*, 28(8):726–728, Aug 2018.